AN INTRODUCTION TO PROGRAMMING WITH C++

SIXTH EDITION



DIANE ZAK

AN INTRODUCTION TO PROGRAMMING WITH C++

This page intentionally left blank

SIXTH EDITION

AN INTRODUCTION TO PROGRAMMING WITH C++

DIANE ZAK

COURSE TECHNOLOGY
CENGAGE Learning™

Australia • Brazil • Japan • Korea • Mexico • Singapore • Spain • United Kingdom • United States



An Introduction to Programming with C++, Sixth Edition
Diane Zak

Executive Editor: Marie Lee
Acquisitions Editor: Amy Jollymore
Freelance Product Manager: Tricia Coia
Senior Content Project Manager: Jill Braiewa
Editorial Assistant: Zina Kresin
Art Director: Marissa Falco
Text Designer: Shawn Girsberger
Print Buyer: Julio Esperas
Proofreader: Andy Smith, Green Pen QA
Indexer: Michael Brackney
Compositor: Integra Software Services

© 2011 Course Technology, Cengage Learning

ALL RIGHTS RESERVED. No part of this work covered by the copyright herein may be reproduced, transmitted, stored or used in any form or by any means—graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act—without the prior written permission of the publisher.

For product information and technology assistance, contact us at
Cengage Learning Customer & Sales Support, 1-800-354-9706

For permission to use material from this text or product, submit all requests online at **cengage.com/permissions**
Further permissions questions can be emailed to
permissionrequest@cengage.com

Library of Congress Control Number: 2009941969

ISBN-13: 978-0-538-46652-3

ISBN-10: 0-538-46652-9

Course Technology
20 Channel Center Street
Boston, MA 02210
USA

Some of the product names and company names used in this book have been used for identification purposes only and may be trademarks or registered trademarks of their respective manufacturers and sellers.

Course Technology, a part of Cengage Learning, reserves the right to revise this publication and make changes from time to time in its content without notice.

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at:
www.cengage.com/global

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

To learn more about Course Technology, visit
www.cengage.com/coursetechnology

Purchase any of our products at your local college store or at our preferred online store: **www.CengageBrain.com**

Printed in the United States of America
1 2 3 4 5 6 7 16 15 14 13 12 11 10

This page intentionally left blank

Brief Contents

	Preface	xiv
CHAPTER 1	An Introduction to Programming	1
CHAPTER 2	Beginning the Problem-Solving Process	22
CHAPTER 3	Variables and Constants	51
CHAPTER 4	Completing the Problem-Solving Process	77
CHAPTER 5	The Selection Structure	119
CHAPTER 6	More on the Selection Structure	163
CHAPTER 7	The Repetition Structure	213
CHAPTER 8	More on the Repetition Structure	264
CHAPTER 9	Value-Returning Functions	308
CHAPTER 10	Void Functions	370
CHAPTER 11	One-Dimensional Arrays	419
CHAPTER 12	Two-Dimensional Arrays	486
CHAPTER 13	Strings	524
CHAPTER 14	Sequential Access Files	582
APPENDIX A	Answers to Mini-Quizzes and Labs	626
APPENDIX B	C++ Keywords	690

APPENDIX C ASCII Codes 691
APPENDIX D How to Use Microsoft Visual C++ 693
APPENDIX E How to Use Dev-C++. 694
APPENDIX F Classes and Objects 695
Index 721

Contents

Preface xiv

CHAPTER 1 An Introduction to Programming 1

Programming a Computer 2
The Programmer's Job 2
Do I Have What It Takes to Be a Programmer? 2
Employment Opportunities 3
A Brief History of Programming Languages 4
Machine Languages 4
Assembly Languages 4
High-Level Languages 4
Control Structures 6
The Sequence Structure 6
The Selection Structure 7
The Repetition Structure 9
Summary 12
Key Terms 12
Review Questions 13
Exercises 15

CHAPTER 2 Beginning the Problem-Solving Process 22

Problem Solving 23
Solving Everyday Problems 23
Creating Computer Solutions to Problems 24
Step 1—Analyze the Problem 25
Step 2—Plan the Algorithm 27
Step 3—Desk-Check the Algorithm 31
The Gas Mileage Problem 34
Summary 43
Key Terms 44
Review Questions 44
Exercises 46

CHAPTER 3 Variables and Constants 51

Beginning Step 4 in the Problem-Solving Process 52
Internal Memory 52

Selecting a Name for a Memory Location 53
Revisiting the Treyson Mobley Problem 54
Selecting a Data Type for a Memory Location 55
How Data Is Stored in Internal Memory 57
Selecting an Initial Value for a Memory Location 60
Declaring a Memory Location 62
Summary 69
Key Terms 69
Review Questions 71
Exercises 73

CHAPTER 4 Completing the Problem-Solving Process 77

Finishing Step 4 in the Problem-Solving Process 78
Getting Data from the Keyboard 79
Displaying Messages on the Computer Screen 81
Arithmetic Operators in C++ 83
Type Conversions in Arithmetic Expressions 84
The `static_cast` Operator 86
Assignment Statements 87
Step 5—Desk-Check the Program 90
Step 6—Evaluate and Modify the Program 92
Arithmetic Assignment Operators 97
Summary 103
Key Terms 105
Review Questions 107
Exercises 109

CHAPTER 5 The Selection Structure 119

Making Decisions 120
Flowcharting a Selection Structure 123
Coding a Selection Structure in C++ 125
Comparison Operators 127
Swapping Numeric Values 128
Displaying the Sum or Difference 130
Logical Operators 132
Using the Truth Tables 134
Calculating Gross Pay 135
Pass/Fail Program 137
Converting a Character to Uppercase or Lowercase 140
Formatting Numeric Output 141
Summary 150
Key Terms 151
Review Questions 152
Exercises 154

CHAPTER 6 More on the Selection Structure 163

Making Decisions 164
Flowcharting a Nested Selection Structure 168
Coding a Nested Selection Structure 170

Logic Errors in Selection Structures 173
First Logic Error: Using a Compound Condition Rather Than a Nested Selection Structure 175
Second Logic Error: Reversing the Outer and Nested Decisions 177
Third Logic Error: Using an Unnecessary Nested Selection Structure 178
Multiple-Alternative Selection Structures 180
The `switch` Statement 183
Summary 196
Key Terms. 196
Review Questions 197
Exercises 200

CHAPTER 7 The Repetition Structure 213

Repeating Program Instructions 214
Using a Pretest Loop to Solve a Real-World Problem 216
Flowcharting a Pretest Loop 219
The `while` Statement 221
Using Counters and Accumulators 224
The Sales Express Program 225
Counter-Controlled Pretest Loops 228
The `for` Statement 231
The Holmes Supply Program. 233
The Colfax Sales Program. 236
Another Version of the Miller Incorporated Program 238
Summary 249
Key Terms 250
Review Questions 251
Exercises 254

CHAPTER 8 More on the Repetition Structure 264

Posttest Loops 265
Flowcharting a Posttest Loop 267
The `do while` Statement 270
Nested Repetition Structures 273
The Asterisks Program 275
The Savings Calculator Program. 283
The `pow` Function 284
Coding the Savings Calculator Program 286
Modifying the Savings Calculator Program 287
Summary 297
Key Terms 298
Review Questions 298
Exercises 300

CHAPTER 9 Value-Returning Functions 308

Functions 309
The Hypotenuse Program 310
Finding the Square Root of a Number 310

The Random Addition Problems Program 313
Generating Random Integers 314
Creating Program-Defined Value-Returning Functions 322
Calling a Function 326
Function Prototypes 330
The Plano Elementary School Program 333
The Area Calculator Program 336
The Scope and Lifetime of a Variable 340
The Bonus Calculator Program. 340
Summary 357
Key Terms 358
Review Questions 359
Exercises 362

CHAPTER 10 Void Functions 370

Void Functions 371
Passing Variables to a Function 376
Reviewing Passing Variables *by Value* 377
Passing Variables *by Reference* 379
The Salary Program 384
Summary 401
Key Terms 402
Review Questions 402
Exercises 406

CHAPTER 11 One-Dimensional Arrays 419

Arrays 420
One-Dimensional Arrays 420
Declaring and Initializing a One-Dimensional Array 422
Entering Data into a One-Dimensional Array 424
Displaying the Contents of a One-Dimensional Array 426
Coding the XYZ Company's Sales Program 427
Passing a One-Dimensional Array to a Function 433
The Moonbucks Coffee Program—Calculating a Total and Average 436
The KL Motors Program—Searching an Array. 439
The Hourly Rate Program—Accessing an Individual Element 442
The Random Numbers Program 444
Sorting the Data Stored in a One-Dimensional Array 454
Parallel One-Dimensional Arrays 461
Summary 473
Key Terms 474
Review Questions 474
Exercises 478

CHAPTER 12 Two-Dimensional Arrays 486

Using Two-Dimensional Arrays. 487
Declaring and Initializing a Two-Dimensional Array 489
Entering Data into a Two-Dimensional Array 491

Displaying the Contents of a Two-Dimensional Array 494
Coding the Caldwell Company's Orders Program 495
Accumulating the Values Stored in a Two-Dimensional Array 498
Searching a Two-Dimensional Array 500
Passing a Two-Dimensional Array to a Function 507
Summary 515
Key Term 516
Review Questions 516
Exercises 517

CHAPTER 13 Strings 524

The `string` Data Type 525
The Creative Sales Program 526
The `getline` Function 527
The `ignore` Function 531
The ZIP Code Program 535
Determining the Number of Characters Contained in a `string` Variable 535
Accessing the Characters Contained in a `string` Variable 538
The Rearranged Name Program 544
Searching the Contents of a `string` Variable 544
The Annual Income Program 547
Removing Characters from a `string` Variable 548
Replacing Characters in a `string` Variable 551
The Social Security Number Program 553
Inserting Characters Within a `string` Variable 554
The Company Name Program 556
Duplicating a Character Within a `string` Variable 557
Concatenating Strings 558
Summary 568
Key Terms 570
Review Questions 570
Exercises 574

CHAPTER 14 Sequential Access Files 582

File Types 583
The CD Collection Program 583
Creating File Objects 585
Opening a Sequential Access File 586
Determining Whether a File Was Opened Successfully 589
Writing Data to a Sequential Access File 590
Reading Information from a Sequential Access File 592
Testing for the End of a Sequential Access File 594
Closing a Sequential Access File 595
Coding the CD Collection Program 596
Summary 614
Key Terms 614
Review Questions 615
Exercises 618

APPENDIX A Answers to Mini-Quizzes and Labs 626
APPENDIX B C++ Keywords 690
APPENDIX C ASCII Codes 691
APPENDIX D How to Use Microsoft Visual C++ 693
APPENDIX E How to Use Dev-C++. 694
APPENDIX F Classes and Objects 695

Object-Oriented Terminology 696
Defining a Class in C++ 697
Instantiating an Object and Referring to a Public Member 700
Example 1—A Class that Contains Public Data Members Only 702
Header Files 704
Example 2—A Class that Contains a Private Data Member and Public Member Methods 706
Example 3—Using a Class that Contains Two Constructors 709
Example 4—A Class that Contains Overloaded Methods 712
Summary 716
Key Terms. 717
Review Questions 718
Exercises 720

Index 721

Preface

An Introduction to Programming with C++, Sixth Edition uses the C++ programming language to teach programming concepts. This book is designed for a beginning programming course. Although the book provides instructions for using the Microsoft® Visual C++® and Dev-C++ compilers, it can be used with most C++ compilers, often with little or no modification.

Organization and Coverage

An Introduction to Programming with C++, Sixth Edition contains 14 chapters and several appendices. In order to provide the most up-to-date instructions for using the Microsoft Visual C++ and Dev-C++ compilers, Appendices D and E are available online. You can obtain the appendices by connecting to the Course Technology Web site (*www.cengage.com/coursetechnology*) and then navigating to the page for this book. In the chapters, students with no previous programming experience learn how to plan and create well-structured programs. By the end of the book, students will have learned how to write programs using the sequence, selection, and repetition structures, as well as how to create and manipulate functions, sequential access files, arrays, strings, classes, and objects.

Approach

An Introduction to Programming with C++, Sixth Edition is distinguished from other textbooks because of its unique approach, which motivates students by demonstrating why they need to learn the concepts and skills presented. Each chapter begins with an introduction to one or more programming concepts. The concepts are illustrated with code examples and sample programs. The sample programs allow the student to observe how the current concept can be used before they are introduced to the next concept. The concepts are taught using standard C++ commands. Following the concept portion in each chapter (except Chapter 1) are five labs: Stop and Analyze, Plan and Create, Modify, Desk-Check, and Debug. Each lab teaches students how to apply the chapter concepts; however, each does so in a different way.

Features

An Introduction to Programming with C++, Sixth Edition is an exceptional textbook because it also includes the following features:

READ THIS BEFORE YOU BEGIN This section is consistent with Course Technology's unequaled commitment to helping instructors introduce

technology into the classroom. Technical considerations and assumptions about hardware, software, and default settings are listed in one place to help instructors save time and eliminate unnecessary aggravation.



LABS Each chapter contains five labs that teach students how to apply the concepts taught in the chapter to real-world problems. In the first lab, which is the Stop and Analyze lab, students are expected to stop and analyze an existing program. Students plan and create a program in the Plan and Create lab, which is the second lab. The third lab is the Modify lab. This lab requires students to modify an existing program. The fourth lab is the Desk-Check lab, in which students follow the logic of a program by desk-checking it. The fifth lab is the Debug lab. This lab gives students an opportunity to find and correct the errors in an existing program.

STANDARD C++ SYNTAX Like the previous edition of the book, this edition uses the standard C++ syntax in the examples, sample programs, and exercises in each chapter.

TIP These notes provide additional information about the current concept. Examples include alternative ways of writing statements, warnings about common mistakes made when using a particular command, and reminders of related concepts learned in previous chapters.

PSEUDOCODE AND FLOWCHARTS Although pseudocode is the primary tool used when planning the programs in each chapter, flowcharts also are provided for many of the programs. If the flowchart is not in the chapter itself, the student is directed to the Cpp6\Chap*xx*\Ch*xx*Flowcharts.pdf file, where *xx* is the chapter number.

MINI-QUIZZES Mini-quizzes are strategically placed to test students' knowledge at various points in each chapter. Answers to the quiz questions are provided in Appendix A, allowing students to determine whether they have mastered the material covered thus far before continuing with the chapter.

SUMMARY A Summary section follows the labs in each chapter. The Summary section recaps the programming concepts and commands covered in the chapter.

KEY TERMS Following the Summary section in each chapter is a listing of the key terms introduced throughout the chapter, along with their definitions.

REVIEW QUESTIONS Review Questions follow the Key Terms section in each chapter. The Review Questions test the students' understanding of what they learned in the chapter.



PAPER AND PENCIL EXERCISES The Review Questions are followed by Pencil and Paper Exercises, which are designated as TRY THIS, MODIFY THIS, INTRODUCTORY, INTERMEDIATE, ADVANCED, and SWAT THE BUGS. The answers to the TRY THIS Exercises are provided at the end of the chapter. The ADVANCED Exercises provide practice in applying cumulative programming knowledge or allow students to explore alternative solutions to programming tasks. The SWAT THE BUGS Exercises provide an opportunity for students to detect and correct errors in one or more lines of code.



COMPUTER EXERCISES The Computer Exercises provide students with additional practice of the skills and concepts they learned in the chapter. The Computer Exercises are designated as TRY THIS, MODIFY THIS, INTRODUCTORY, INTERMEDIATE, ADVANCED, and SWAT THE BUGS. The answers to the TRY THIS Exercises are provided at the end of the chapter. The ADVANCED Exercises provide practice in applying cumulative programming knowledge or allow students to explore alternative solutions to programming tasks. The SWAT THE BUGS Exercises provide an opportunity for students to detect and correct errors in an existing program.

New to This Edition!

STD NAMESPACE Rather than including a `using` directive for each standard object used in a program, all programs now contain the `using namespace std;` directive.

STRING CLASS The `string` class is now covered in Chapter 13. In the chapter, students learn how to declare and utilize `string` variables and `string` named constants in a program. They also learn how to concatenate strings and use many of the functions available in the `string` class.

CHAPTERS 3 AND 4 Chapters 3 and 4 from the previous edition of the book have been redesigned to make the material easier for students to comprehend. Chapter 3 now covers only variables and named constants, which are challenging concepts for beginner programmers. Chapter 4 shows the student how to get numeric and character input from the keyboard, write assignment statements, and display information on the computer screen. Chapter 4 also covers the last three steps in the problem-solving process.

ARRAYS One-dimensional arrays and two-dimensional arrays are now covered in separate chapters. One-dimensional arrays are covered in Chapter 11, and two-dimensional arrays are covered in Chapter 12.

APPENDIX A The answers to both the Mini-Quiz questions and the Labs are now located in one convenient place in the book: Appendix A.

APPENDICES D AND E Appendix D contains the instructions for using the Microsoft Visual C++ compiler, and Appendix E contains the instructions for using the Dev-C++ compiler. In order to provide the most up-to-date instructions for using both compilers, Appendices D and E are available online. You can obtain the appendices by connecting to the Course Technology Web site (*www.cengage.com/coursetechnology*) and then navigating to the page for this book.

APPENDIX F Appendix F covers Classes and Objects. This topic was originally covered in Chapter 14 in the previous edition of the book.

NEW EXAMPLES, SAMPLE PROGRAMS, AND EXERCISES Each chapter has been updated with new examples, sample programs, and exercises.

NEW FORMAT The book has a new, more convenient format. All of the questions and exercises are now located at the end of the chapter.

Instructor Resources and Supplements

All of the resources available with this book are provided to the instructor on a single CD-ROM. Many also can be found on the Course Technology Web site (*www.cengage.com/coursetechnology*).

ELECTRONIC INSTRUCTOR'S MANUAL The Instructor's Manual that accompanies this textbook includes additional instructional material to assist in class preparation, including Sample Syllabi, Chapter Outlines, Technical Notes, Lecture Notes, Quick Quizzes, Teaching Tips, Discussion Topics, and Additional Case Projects.

EXAMVIEW® This textbook is accompanied by ExamView, a powerful testing software package that allows instructors to create and administer printed, computer (LAN-based), and Internet exams. ExamView includes hundreds of questions that correspond to the topics covered in this text, enabling students to generate detailed study guides that include page references for further review. The computer-based and Internet testing components allow students to take exams at their computers, and also save the instructor time by grading each exam automatically.

MICROSOFT® POWERPOINT® PRESENTATIONS This book offers Microsoft PowerPoint slides for each chapter. These are included as a teaching aid for classroom presentation, to make available to students on the network for chapter review, or to be printed for classroom distribution. Instructors can add their own slides for additional topics they introduce to the class.

DATA FILES Data Files are necessary for completing the Labs and Computer Exercises in this book. The Data Files are provided on the Instructor Resources CD-ROM and also may be found on the Course Technology Web site at *www.cengage.com/coursetechnology*.

SOLUTION FILES Solutions to the Labs, Review Questions, Pencil and Paper Exercises, and Computer Exercises are provided on the Instructor Resources CD-ROM and also may be found on the Course Technology Web site at *www.cengage.com/coursetechnology*. The solutions are password protected.

FIGURE FILES The sample programs that appear in the figures throughout the book are provided on the Instructor Resources CD-ROM.

DISTANCE LEARNING Course Technology offers online WebCT and Blackboard courses for this text to provide the most complete and dynamic learning experience possible. When you add online content to one of your courses, you're adding a lot: automated tests, topic reviews, quick quizzes, and additional case projects with solutions. For more information on how to bring distance learning to your course, contact your local Course Technology sales representative.

Acknowledgments

Writing a book is a team effort rather than an individual one. I would like to take this opportunity to thank my team, especially Tricia Coia (Freelance Product Manager), Jill Braiewa (Senior Content Project Manager), Sreejith Govindan (Full Service Project Manager), and Nicole Ashton (Quality Assurance). Thank you for your support, enthusiasm, patience, and hard work; it made a difficult task much easier. Last, but certainly not least, I want to thank Matthew Alimagham (Spartanburg Technical College) and Linda Cohen (Forsyth Technical Community College) for their invaluable ideas and comments. And an extra special thank you to Bill Tucker (Austin Community College) for going way above and beyond to help me on this project. Your attention to detail and your willingness to share your ideas and your experiences with the previous edition of the book were very much appreciated.

Diane Zak

Got a Job in Computing . . . ?

We hope you enjoyed the Q&A on the inside front cover of this book. If you'd like to suggest that we interview someone you know, a recent graduate who has landed an interesting job in computing, please send your suggestions via e-mail to Amy Jollymore, Acquisitions Editor, at Amy.Jollymore@Cengage.com.

Read This Before You Begin

Technical Information

Data Files

You will need data files to complete the Labs and Computer Exercises in this book. Your instructor may provide the data files to you. You may obtain the files electronically on the Course Technology Web site (*www.cengage.com/coursetechnology*).

Each chapter in this book has its own set of data files, which are stored in a separate folder within the Cpp6 folder. The files for Chapter 4 are stored in the Cpp6\Chap04 folder. Similarly, the files for Chapter 5 are stored in the Cpp6\Chap05 folder. Throughout this book, you will be instructed to open files from or save files to these folders.

You can use a computer in your school lab or your own computer to complete the Labs and Computer Exercises in this book.

Using Your Own Computer

To use your own computer to complete the Labs and Computer Exercises in this book, you will need a C++ compiler. The book was written and Quality Assurance tested using Microsoft Visual C++ 2010. It also was tested using Dev-C++. However, the book can be used with most C++ compilers, often with little or no modification. If your book came with a copy of Microsoft Visual C++, then you may install that on your computer and use it to complete the material.

Visit Our Web Site

Additional materials designed for this textbook might be available through the Course Technology Web site, *www.cengage.com/coursetechnology*. Search this site for more details.

To the Instructor

To complete the Labs and Computer Exercises in this book, your students must use a set of data files. These files are included on the Instructor Resources CD-ROM. They may also be obtained electronically through the Course Technology Web site at *www.cengage.com/coursetechnology*. Follow the instructions in the Help file to copy the data files to your server or standalone computer. You can view the Help file using a text editor such as WordPad or Notepad. Once the files are copied, you should instruct your users how to copy the files to their own computers or workstations.

The material in this book was written and Quality Assurance tested using Microsoft Visual C++ 2010. It also was tested using Dev-C++. However, the book can be used with most C++ compilers, often with little or no modification.

Course Technology Data Files

You are granted a license to copy the data files to any computer or computer network used by individuals who have purchased this book.

CHAPTER 1

An Introduction to Programming

After studying Chapter 1, you should be able to:

- Define the terminology used in programming
- Explain the tasks performed by a programmer
- Describe the qualities of a good programmer
- Understand the employment opportunities for programmers and software engineers
- Explain the history of programming languages
- Explain the sequence, selection, and repetition structures
- Write simple algorithms using the sequence, selection, and repetition structures

Programming a Computer

In essence, the word **programming** means *giving a mechanism the directions to accomplish a task*. If you are like most people, you've already programmed several mechanisms. For example, at one time or another, you probably programmed your digital video recorder (DVR) in order to schedule a timed-recording of a movie. You also may have programmed the speed dial feature on your cell phone. Or you may have programmed your coffee maker to begin the brewing process before you wake up in the morning. Like your DVR, cell phone, and coffee maker, a computer also is a mechanism that can be programmed. The directions given to a computer are called **computer programs** or, more simply, **programs**. The people who write programs are called **programmers**. Programmers use a variety of special languages, called **programming languages**, to communicate with the computer. Some popular programming languages are C++, Visual Basic, C#, and Java. In this book, you will use the C++ programming language.

The Programmer's Job

When a company has a problem that requires a computer solution, typically it is a programmer that comes to the rescue. The programmer might be an employee of the company; or he or she might be a freelance programmer, which is a programmer who works on temporary contracts rather than for a long-term employer. First, the programmer meets with the user, which is the person (or persons) responsible for describing the problem. In many cases, this person or persons also will eventually use the solution. Depending on the complexity of the problem, multiple programmers may be involved, and they may need to meet with the user several times. The purpose of the initial meetings is to determine the exact problem and to agree on the desired solution. After the programmer and user agree on the solution, the programmer begins converting the solution into a computer program. During the conversion phase, the programmer meets periodically with the user to determine whether the program fulfills the user's needs and to refine any details of the solution. When the user is satisfied that the program does what he or she wants it to do, the programmer rigorously tests the program with sample data before releasing it to the user. In many cases, the programmer also provides the user with a manual that explains how to use the program. As this process indicates, the creation of a good computer solution to a problem—in other words, the creation of a good program—requires a great deal of interaction between the programmer and the user.

Do I Have What It Takes to Be a Programmer?

According to the 2008–09 Edition of the Occupational Outlook Handbook (OOH), published by the U.S. Department of Labor's Bureau of Labor Statistics, "When hiring programmers, employers look for people with the necessary programming skills who can think logically and pay close attention to detail. Programming calls for patience, persistence, and the ability to work on exacting analytical work, especially under pressure. Ingenuity and creativity also are particularly important when programmers design solutions and test their work for potential failures. . . . Because programmers are

expected to work in teams and interact directly with users, employers want programmers who are able to communicate with nontechnical personnel. Business skills are also important, especially for those wishing to advance to managerial positions." If this description sounds like you, then you probably have what it takes to be a programmer. But if it doesn't sound like you, it's still worth your time to understand the programming process, especially if you are planning a career in business. Knowing even a little bit about the programming process will allow you, the manager of a department, to better communicate your department's needs to a programmer. It also will give you the confidence to question the programmer when he claims that he can't make the program modification you requested. In addition, it will help you determine whether the $15,000 quote you received from a freelance programmer seems reasonable. Lastly, understanding the process a computer programmer follows when solving a problem can help you solve problems that don't require a computer solution.



Programming teams often contain subject matter experts, who may or may not be programmers. For example, an accountant might be part of a team working on a program that requires accounting expertise.

Employment Opportunities

But if, after reading this book, you are excited about the idea of working as a computer programmer, here is some information on employment opportunities. When searching for a job in computer programming, you will encounter ads for "computer programmers" as well as for "computer software engineers." Although job titles and descriptions vary, computer software engineers typically are responsible for designing an appropriate solution to a user's problem, while computer programmers are responsible for translating the solution into a language that the computer can understand. The process of translating the solution is called **coding**. Keep in mind that, depending on the employer and the size and complexity of the user's problem, the design and coding tasks may be performed by the same employee, no matter what his or her job title is. In other words, it's not unusual for a software engineer to code her solution, just as it's not unusual for a programmer to have designed the solution he is coding. Typically, computer software engineers are expected to have at least a bachelor's degree in computer engineering or computer science, along with practical work experience. Computer programmers usually need at least an associate's degree in computer science, mathematics, or information systems, as well as proficiency in one or more programming languages. Computer programmers and software engineers are employed in almost every industry, such as telecommunications companies, software publishers, financial institutions, insurance carriers, educational institutions, and government agencies. According to the May 2008 Occupational Employment Statistics, programmers held about 394,230 jobs and had a mean annual wage of $73,470. Software engineers, on the other hand, held about 494,160 jobs with a mean annual wage of $87,900. The Bureau of Labor Statistics predicts that employment of programmers will decline slowly, decreasing by 4% from 2006 to 2016. However, the employment of computer software engineers is projected to increase by 38% over the same period. There is a great deal of competition for programming and software engineering jobs, so jobseekers will need to keep up to date with the latest programming languages and technologies. More information about computer programmers and computer software engineers can be found on the Bureau of Labor Statistics Web site at *www.bls.gov*.

A Brief History of Programming Languages

Just as human beings communicate with each other through the use of languages such as English, Spanish, Hindi, and Chinese, programmers use a variety of programming languages to communicate with the computer. In the next sections, you will follow the progression of programming languages from machine languages to assembly languages, and then to high-level languages.

Machine Languages

Within a computer, all data is represented by microscopic electronic switches that can be either off or on. The off switch is designated by a 0, and the on switch is designated by a 1. Because computers can understand only these on and off switches, the first programmers had to write the program instructions using nothing but combinations of 0s and 1s; for example, a program might contain the instruction `00101 10001 10000`. Instructions written in 0s and 1s are called **machine language** or **machine code**. The machine languages (each type of machine has its own language) represent the only way to communicate directly with the computer. As you can imagine, programming in machine language is very tedious and error-prone and requires highly trained programmers.

Assembly Languages

Slightly more advanced programming languages are called assembly languages. The **assembly languages** simplify the programmer's job by allowing the programmer to use mnemonics in place of the 0s and 1s in the program. **Mnemonics** are memory aids—in this case, alphabetic abbreviations for instructions. For example, most assembly languages use the mnemonic `ADD` to represent an add operation and the mnemonic `MUL` to represent a multiply operation. An example of an instruction written in an assembly language is `ADD bx, ax`. Programs written in an assembly language require an **assembler**, which also is a program, to convert the assembly instructions into machine code—the 0s and 1s the computer can understand. Although it is much easier to write programs in assembly language than in machine language, programming in assembly language still is tedious and requires highly trained programmers. Programs written in assembly language are machine specific and usually must be rewritten in a different assembly language to run on a different computer.

High-Level Languages

High-level languages represent the next major development in programming languages. **High-level languages** are a vast improvement over machine and assembly languages, because they allow the programmer to use instructions that more closely resemble the English language. An example of an instruction written in a high-level language is `grossPay = hours * rate`. In addition, high-level languages are more machine independent than are machine and assembly languages. As a result, programs written in a high-level language can be used on many different types of computers. Programs written in a high-level language usually require a compiler, which also is a

program, to convert the English-like instructions into the 0s and 1s the computer can understand. Some high-level languages also offer an additional program called an interpreter. Unlike a **compiler**, which translates all of a program's high-level instructions before running the program, an **interpreter** translates the instructions line by line as the program is running.

Like their predecessors, the first high-level languages were used to create procedure-oriented programs. When writing a **procedure-oriented program**, the programmer concentrates on the major tasks that the program needs to perform. A payroll program, for example, typically performs several major tasks, such as inputting the employee data, calculating the gross pay, calculating the taxes, calculating the net pay, and outputting a paycheck. The programmer must instruct the computer every step of the way, from the start of the task to its completion. In a procedure-oriented program, the programmer determines and controls the order in which the computer processes the instructions. In other words, the programmer must determine not only the proper instructions to give the computer, but the correct sequence of those instructions as well. Examples of high-level languages used to create procedure-oriented programs include COBOL (Common Business Oriented Language), BASIC (Beginner's All-Purpose Symbolic Instruction Code), and C.

More advanced high-level languages can be used to create object-oriented programs in addition to procedure-oriented ones. Different from a procedure-oriented program, which focuses on the individual tasks the program must perform, an **object-oriented program** requires the programmer to focus on the objects that the program can use to accomplish its goal. The objects can take on many different forms. For example, programs written for the Windows environment typically use objects such as check boxes, list boxes, and buttons. A payroll program, on the other hand, might utilize objects found in the real world, such as a time card object, an employee object, or a check object. Because each object is viewed as an independent unit, an object can be used in more than one program, usually with little or no modification. A check object used in a payroll program, for example, also can be used in a sales revenue program (which receives checks from customers) and an accounts payable program (which issues checks to creditors). The ability to use an object for more than one purpose enables code-reuse, which saves programming time and money—an advantage that contributes to the popularity of object-oriented programming. Examples of high-level languages that can be used to create both procedure-oriented and object-oriented programs include C++, Visual Basic, Java, and C#. In this book, you will learn how to use the C++ programming language to create procedure-oriented and object-oriented programs.



Most objects in an object-oriented program are designed to perform multiple tasks. These tasks are programmed using the same techniques used in procedure-oriented programming.

Mini-Quiz 1-1



The answers to Mini-Quiz questions are located in Appendix A.

1. Instructions written in 0s and 1s are called _____ language.

2. When writing _____ program, the programmer concentrates on the major tasks needed to accomplish a goal.

   a. a procedure-oriented

   b. an object-oriented

3. When writing _____ program, the programmer breaks up a problem into interacting objects.

   a. a procedure-oriented

   b. an object-oriented

4. Most high-level languages use a _____ to translate the instructions into a language that the computer can understand.

Control Structures

All computer programs, no matter how simple or how complex, are written using one or more of three basic structures: sequence, selection, and repetition. These structures are called **control structures** or **logic structures**, because they control the flow of a program's logic. You will use the sequence structure in every program you write. In most programs, you also will use the selection and repetition structures. This chapter gives you an introduction to the three control structures. It also introduces you to a robot named Robin, who will help illustrate the control structures. More detailed information about each structure, as well as how to implement these structures using the C++ language, is provided in subsequent chapters.

The Sequence Structure

You already are familiar with the sequence structure, because you use it each time you follow a set of directions, in order, from beginning to end. A cookie recipe, for instance, provides a good example of the sequence structure. To get to the finished product (edible cookies), you need to follow each recipe instruction in order, beginning with the first instruction and ending with the last. Likewise, the **sequence structure** in a computer program directs the computer to process the program instructions, one after another, in the order listed in the program. You will find the sequence structure in every program.

You can observe how the sequence structure works by programming Robin, the robot. Like a computer, Robin has a limited instruction set. In other words, she can understand only a specific number of instructions, also called commands. For now, you will use only two of the commands from Robin's instruction set: walk forward and open the bedroom door. When told to walk forward, Robin takes one complete step forward. In other words, she moves her right foot forward one step and then moves her left foot to meet her right foot. For this first example, Robin is standing in her hallway facing her bedroom door. The door, which is closed, is two steps away from Robin. Your task is to write the instructions, using only the commands that Robin understands, that direct Robin to enter her bedroom. Figure 1-1 shows the problem specification along with an illustration of the problem. It also shows the instructions that will get Robin inside her bedroom. The four instructions shown in the figure are called an **algorithm**, which is a set of step-by-step instructions that accomplish a task. For Robin to enter her bedroom, she must follow the instructions in order—in other words, in sequence.

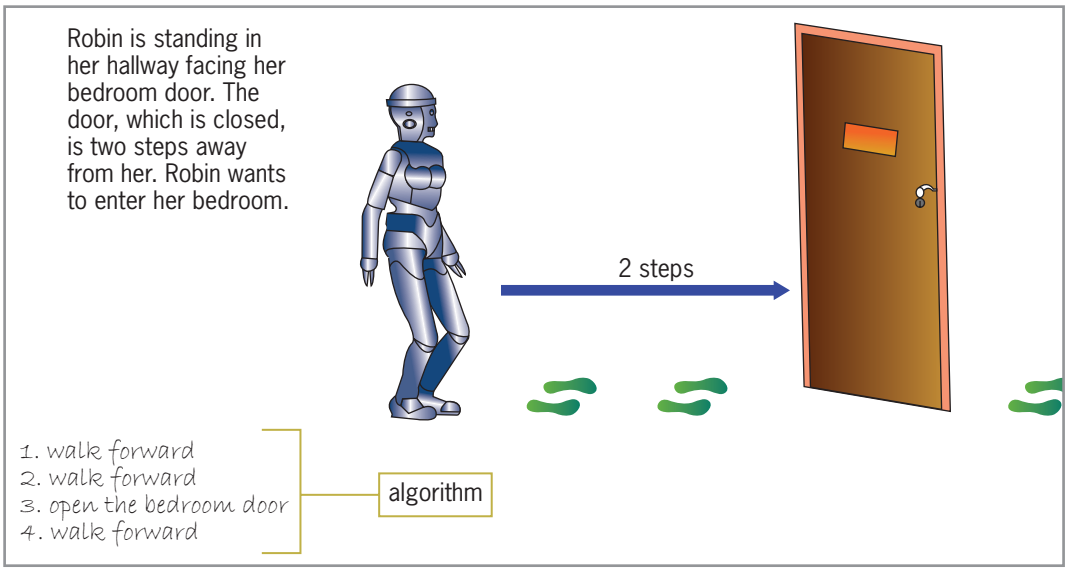

Robin is standing in her hallway facing her bedroom door. The door, which is closed, is two steps away from her. Robin wants to enter her bedroom.
2 steps
1. walk forward
2. walk forward
3. open the bedroom door
4. walk forward
algorithm

Figure 1-1 An example of the sequence structure

The Selection Structure

As with the sequence structure, you already are familiar with the **selection structure**, also called the **decision structure**. The selection structure indicates that a decision (based on some condition) needs to be made, followed by an appropriate action derived from that decision. You use the selection structure every time you drive your car and approach a railroad crossing. Your decision, as well as the appropriate action, is based on whether the crossing signals (flashing lights and ringing bells) are on or off. If the crossing signals are on, you stop your car before crossing the railroad tracks; otherwise, you proceed with caution over the railroad tracks. When used in a computer program, the selection structure alerts the computer that a decision needs to be made, and it provides the appropriate action to take based on the result of that decision.

To observe how the selection structure works, we'll make a slight change to the problem specification shown in Figure 1-1. This time, Robin's bedroom door may or may not be closed. What changes will need to be made to the original algorithm from Figure 1-1 as a result of this minor modification? The first two instructions in the original algorithm position Robin in front of her bedroom door; Robin will still need to follow those instructions. The third instruction tells Robin to open the bedroom door. That instruction was correct for the original problem specification, which states that the bedroom door is closed. However, in the modified problem specification, the status of the bedroom door is not known: it could be closed or it could already be open. As a result, Robin will need to make a decision and then take the appropriate action based on the result. More specifically, Robin will need to determine whether the bedroom door is closed and then open the door only if it needs to be opened. To write an algorithm to accomplish the current task, you need to use two additional instructions from Robin's instruction set: *if (the bedroom door is closed)* and *end if*. The *if (the bedroom door is closed)* instruction allows Robin to make a decision about the status of the bedroom door, and it represents the beginning of a selection

structure. The portion within the parentheses is called the condition and specifies the decision that Robin must make. Notice that the condition results in either a true or false answer: either the bedroom door is closed (true) or it's not closed (false). The *end if* instruction denotes the end of a selection structure. The last instruction in the original algorithm positions Robin one step inside her bedroom; Robin will still need to follow that instruction. Figure 1-2 shows the modified problem specification along with the modified algorithm. Notice that the *open the bedroom door* instruction is indented within the selection structure. Indenting in this manner indicates that the instruction should be followed only when the bedroom door is closed—in other words, only when the condition results in an answer of true. The instructions to be followed when a selection structure's condition evaluates to true are referred to as the structure's true path. Although the true path in Figure 1-2 includes only one instruction, it can include many instructions.

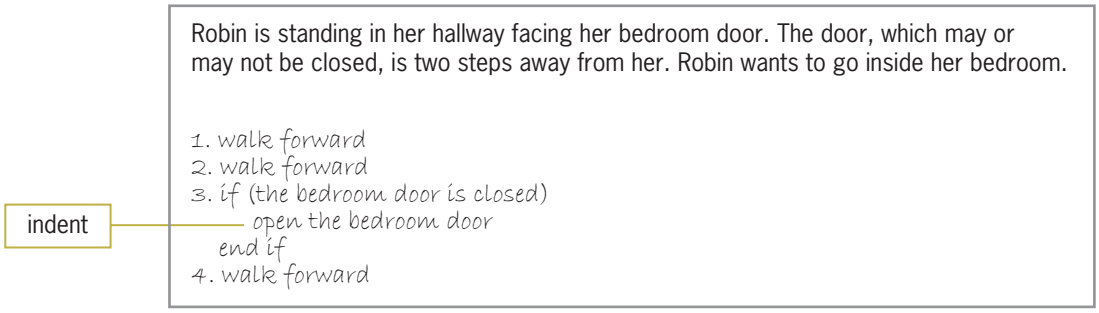

Robin is standing in her hallway facing her bedroom door. The door, which may or may not be closed, is two steps away from her. Robin wants to go inside her bedroom.
1. walk forward
2. walk forward
3. if (the bedroom door is closed)
open the bedroom door
end if
4. walk forward
indent

Figure 1-2 An example of the selection structure

Figure 1-3 shows another example of the selection structure. In this example, Robin is holding either a red or yellow balloon, and she is facing two boxes that are located five steps away from her. One of the boxes is colored yellow, and the other is colored red. Your task is to instruct Robin to drop the balloon into the appropriate box: the yellow balloon belongs in the yellow box, and the red balloon belongs in the red box. To write an algorithm to accomplish the current task, you need to use four additional instructions from Robin's instruction set: *if (the balloon is red)*, *else*, *drop the balloon in the red box*, and *drop the balloon in the yellow box*. The additional instructions allow Robin to make a decision about the color of the balloon she is holding and then take the appropriate action based on that decision. Figure 1-3 shows the problem specification along with an illustration of the problem. It also shows an algorithm that will solve the problem. Unlike the selection structure from Figure 1-2, which requires Robin to take a specific action only when the structure's condition evaluates to true, the selection structure in Figure 1-3 requires her to take one action when the condition evaluates to true, but a different action when it evaluates to false. In other words, the selection structure in Figure 1-3 has both a true path and a false path. The *else* instruction marks the beginning of the false path instructions. Notice that the *drop the balloon in the red box* and *drop the balloon in the yellow box* instructions are indented within their respective paths. Indenting in this manner clearly indicates the instruction to be followed when the condition evaluates to true (the balloon is red), as well as the one to be followed when

the condition evaluates to false (the balloon is not red). Although both paths in Figure 1-3's selection structure contain only one instruction, each can contain many instructions.

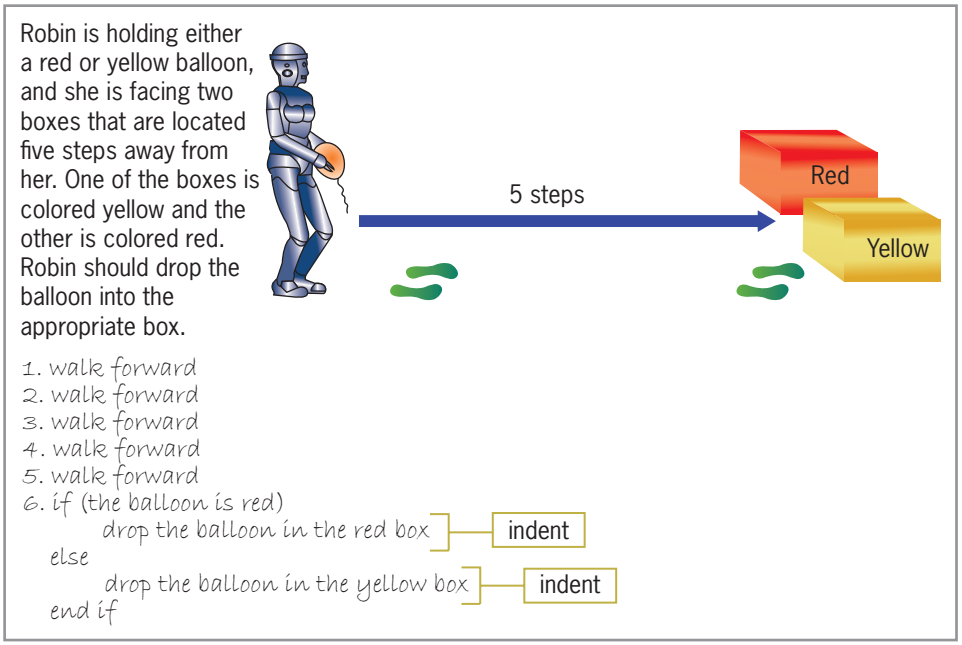

Robin is holding either a red or yellow balloon, and she is facing two boxes that are located five steps away from her. One of the boxes is colored yellow and the other is colored red. Robin should drop the balloon into the appropriate box.
5 steps
Red
Yellow
1. walk forward
2. walk forward
3. walk forward
4. walk forward
5. walk forward
6. if (the balloon is red)
drop the balloon in the red box
indent
else
drop the balloon in the yellow box
indent
end if

Figure 1-3 Another example of the selection structure

The Repetition Structure

The last of the three control structures is the repetition structure. Like the sequence and selection structures, you already are familiar with the repetition structure. For example, shampoo bottles typically include the repetition structure in the directions for washing your hair. Those directions usually tell you to repeat the "apply shampoo to hair," "lather," and "rinse" steps until your hair is clean. When used in a program, the **repetition structure** directs the computer to repeat one or more instructions until some condition is met, at which time the computer should stop repeating the instructions. The repetition structure also is referred to as a **loop** or as **iteration**.

You can use the repetition structure to simplify the algorithm shown in Figure 1-3. To do this, you will need to use two additional instructions from Robin's instruction set: *repeat x times* (where *x* is the number of times you want Robin to repeat something) and *end repeat*. The *repeat x times* and *end repeat* instructions indicate the beginning and end, respectively, of a repetition structure. The modified algorithm, which contains two steps rather than six steps, is shown in Figure 1-4. Notice that the five *walk forward* instructions are replaced by a repetition structure that simply directs Robin to repeat the *walk forward* instruction five times. Also notice that the instruction to be repeated is indented within the repetition structure. Indenting in this manner indicates that the instruction is part of the repetition structure and, therefore, needs to be repeated the specified number of times. Although the repetition structure in Figure 1-4 includes only one instruction, a repetition structure can include many instructions.

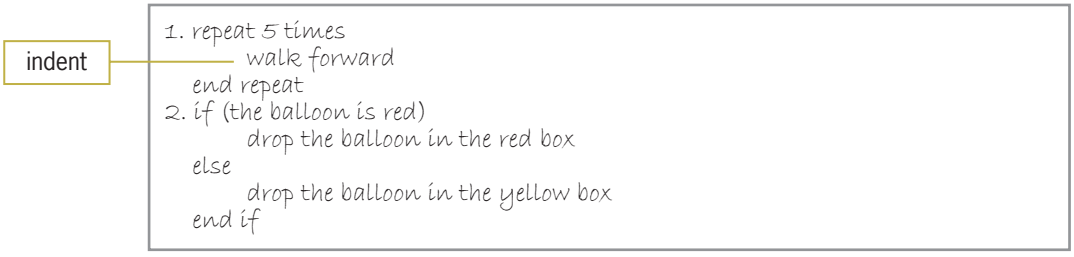


Figure 1-4 Modified algorithm showing the repetition structure

The algorithm shown in Figure 1-4 will work only if Robin is five steps away from the boxes. But what if you don't know precisely how many steps separate Robin from the boxes? In that case, you need to replace the *repeat 5 times* instruction with another instruction from Robin's instruction set. That instruction is *repeat until you are directly in front of the boxes*. The new algorithm with the modified condition in the repetition structure is shown in Figure 1-5. The repetition structure tells Robin to keep walking forward until she is directly in front of the boxes. Depending on the number of steps between Robin and the boxes, Robin may need to walk forward 0 times, 5 times, 10 times, or even 500 times before evaluating the selection structure's condition.

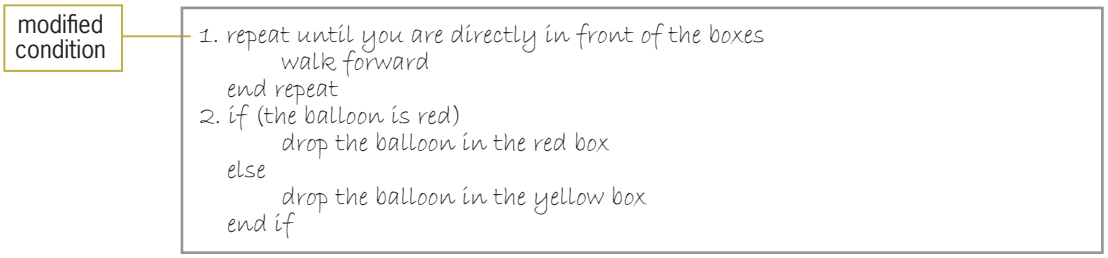

modified condition
1. repeat until you are directly in front of the boxes
walk forward
end repeat
2. if (the balloon is red)
drop the balloon in the red box
else
drop the balloon in the yellow box
end if

Figure 1-5 Algorithm showing the modified condition in the repetition structure



The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 1-2

1. The three basic control structures are _____, _____, and _____.
2. All programs contain the _____ structure.
3. The step-by-step instructions that accomplish a task are called a(n) _____.
4. You use the _____ structure to repeat one or more instructions in a program.
5. The _____ structure ends when its condition has been met.
6. The _____ structure, also called the decision structure, instructs the computer to make a decision and then take some action based on the result of the decision.



LAB 1-1 Stop and Analyze



The answers to the labs are located in Appendix A.

A local business employs five salespeople and pays a 3% bonus on a salesperson's sales. Your task is to create a program that calculates the amount of each salesperson's bonus. The program should print each salesperson's name and bonus amount. Study the algorithm shown in Figure 1-6 and then answer the questions.

```
repeat 5 times
        enter the salesperson's name and sales amount
        calculate the bonus amount by multiplying the sales amount by 3%
        print the salesperson's name and bonus amount
end repeat
```

Figure 1-6 Algorithm for Lab 1-1

QUESTIONS

1. Which control structures are used in the algorithm shown in Figure 1-6?
2. What will the algorithm shown in Figure 1-6 print when the user enters Mary Smith and 2000 as the salesperson's name and sales amount, respectively?
3. How would you modify the algorithm shown in Figure 1-6 so that it also prints the salesperson's sales amount?
4. How would you modify the algorithm shown in Figure 1-6 so that it can be used for any number of salespeople?
5. How would you modify the algorithm shown in Figure 1-6 so that it allows the user to enter the bonus rate and then uses that rate to calculate the five bonus amounts?



LAB 1-2 Plan and Create

Using only the instructions shown in Figure 1-7, create an algorithm that shows the steps an instructor takes when grading a test that contains 25 questions.

```
end if
end repeat
If (the student's answer is not the same as the correct answer)
mark the student's answer incorrect
read the student's answer and the correct answer
repeat 25 times
```

Figure 1-7 Instructions for Lab 1-2



LAB 1-3 Modify

Modify the algorithm shown in Figure 1-6 so that it gives a 3.5% bonus to salespeople selling more than $2,000. All other salespeople should receive a 3% bonus.

Summary

- Programs are the step-by-step instructions that tell a computer how to perform a task. Programmers, the people who write computer programs, use various programming languages to communicate with the computer. The first programming languages were machine languages, also called machine code. The assembly languages came next, followed by the high-level languages. The first high-level languages were used to create procedure-oriented programs. More advanced high-level languages are used to create object-oriented programs, as well as procedure-oriented ones.
- An algorithm is the set of step-by-step instructions that accomplish a task. The algorithms for all computer programs contain one or more of the following three control structures: sequence, selection, and repetition. The control structures, also called logic structures, are so named because they control the flow of a program's logic.
- The sequence structure directs the computer to process the program instructions, one after another, in the order listed in the program. The selection structure, also called the decision structure, directs the computer to make a decision and then select an appropriate action based on that decision. The repetition structure directs the computer to repeat one or more program instructions until some condition is met. The sequence structure is used in all programs. Most programs also contain both the selection and repetition structures.

Key Terms

Algorithm—the set of step-by-step instructions that accomplish a task

Assembler—a program that converts assembly instructions into machine code

Assembly languages—programming languages that use mnemonics, such as ADD

Coding—the process of translating a solution into a language that the computer can understand

Compiler—a program that converts high-level instructions into a language that the computer can understand; unlike an interpreter, a compiler converts all of a program's instructions before running the program

Computer programs—the directions given to computers; also called programs

Control structures—the structures that control the flow of a program's logic; also called logic structures; sequence, selection, and repetition

Decision structure—another term for the selection structure

High-level languages—programming languages whose instructions more closely resemble the English language

Interpreter—a program that converts high-level instructions into a language that the computer can understand; unlike a compiler, an interpreter converts a program's instructions, line by line, as the program is running

Iteration—another term for the repetition structure

Logic structures—another term for control structures

Loop—another term for the repetition structure

Machine code—another term for machine language

Machine language—computer instructions written in 0s and 1s; also called machine code

Mnemonics—the alphabetic abbreviations used to represent instructions in assembly languages

Object-oriented program—a program designed by focusing on the objects that the program could use to accomplish its goal

Procedure-oriented program—a program designed by focusing on the individual tasks to be performed

Programmers—the people who write computer programs

Programming—giving a mechanism the directions to accomplish a task

Programming languages—languages used to communicate with a computer

Programs—the directions given to computers; also called computer programs

Repetition structure—the control structure that directs the computer to repeat one or more instructions until some condition is met, at which time the computer should stop repeating the instructions; also called a loop or iteration

Selection structure—the control structure that directs the computer to make a decision and then take the appropriate action based on that decision; also called the decision structure

Sequence structure—the control structure that directs the computer to process each instruction in the order listed in the program

Review Questions

1. Which of the following is not a programming control structure?
   a. repetition
   b. selection
   c. sequence
   d. sorting

2. Which of the following control structures is used in every program?
   a. repetition
   b. selection
   c. sequence
   d. switching

3. The set of instructions for adding together two numbers is an example of the _____ structure.
   a. control
   b. repetition
   c. selection
   d. sequence

4. The set of step-by-step instructions that solve a problem is called _____.
   a. an algorithm
   b. a list
   c. a plan
   d. a sequential structure

5. The recipe instruction "Beat until smooth" is an example of the _____ structure.
   a. control
   b. repetition
   c. selection
   d. sequence

6. The instruction "If it's raining outside, take an umbrella to work" is an example of the _____ structure.
   a. control
   b. repetition
   c. selection
   d. sequence

7. Which control structure would an algorithm use to determine whether a credit card holder is over his credit limit?
   a. repetition
   b. selection
   c. both repetition and selection

8. Which control structure would an algorithm use to calculate a 5% commission for each of a company's salespeople?
   a. repetition
   b. selection
   c. both repetition and selection

9. A company pays a 3% annual bonus to employees who have been with the company more than 5 years; other employees receive a 1% bonus. Which control structure would an algorithm use to calculate every employee's bonus?
   a. repetition
   b. selection
   c. both repetition and selection

10. Which control structure would an algorithm use to determine whether a customer is entitled to a senior discount?
    a. repetition
    b. selection
    c. both repetition and selection

Exercises



Pencil and Paper

You will use Robin (the robot) to complete Pencil and Paper Exercises 1, 3, 4, and 7. Robin's instruction set is shown in Figure 1-8.

```
drop the toy in the toy chest
else
end if
end repeat
if (the box is red)
if (the flower is white)
jump over the box
pick the flower with your left hand
pick the flower with your right hand
repeat x times
repeat until you are directly in front of the chair
repeat until you are directly in front of the toy chest
sit down
throw the box out of the way
turn right 90 degrees
walk forward
```

Figure 1-8

TRY THIS

1. As illustrated in Figure 1-9, Robin is five steps away from a box, which is an unknown distance away from a chair. Using only the instructions listed in Figure 1-8, create an algorithm that directs Robin to jump over the box and sit in the chair. Be sure to indent the instructions appropriately. (The answers to TRY THIS Exercises are located at the end of the chapter.)

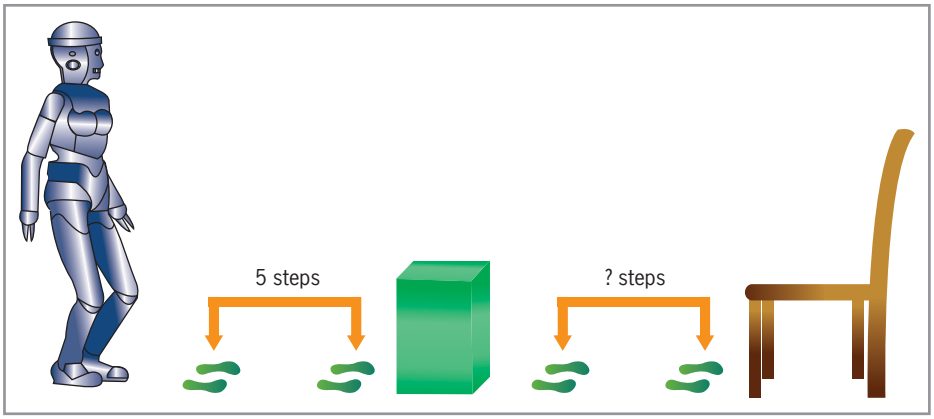


Figure 1-9

TRY THIS

2. A store gives a 10% discount to customers who are at least 65 years old. Using only the instructions shown in Figure 1-10, write an algorithm that prints the amount of money a customer owes. Be sure to indent the instructions appropriately. (The answers to TRY THIS Exercises are located at the end of the chapter.)

```
calculate the amount due by multiplying the amount due by 90%
enter the customer's age and the amount due
if (the customer's age is greater than or equal to 65)
end if
print the amount due
```

Figure 1-10

MODIFY THIS

3. Using only the instructions shown earlier in Figure 1-8, modify the answer to TRY THIS Exercise 1 as follows: Robin must jump over the box if the box is red; otherwise, she must throw the box out of the way.

INTRODUCTORY

4. Robin is facing a toy chest that is zero or more steps away from her. She is carrying a toy in her right hand. Using only the instructions shown earlier in Figure 1-8, create an algorithm that directs Robin to drop the toy in the toy chest. Be sure to indent the instructions appropriately.

INTRODUCTORY

5. You have just purchased a new personal computer system. Before putting the system components together, you read the instruction booklet that came with the system. The booklet contains a list of the components that you should have received. The booklet advises you to verify that you received all of the components by matching those that you received with those on the list. If a component was received, you should cross its name off the list; otherwise, you should draw a circle around the component's name in the list. Using only the instructions listed in Figure 1-11, create an algorithm that shows the steps you should take to verify that you received the correct components. Be sure to indent the instructions appropriately.

```
cross the component name off the list
read the component name from the list
circle the component's name on the list
search for the component
if (the component was received)
else
repeat for each component name on the list
end if
end repeat
```

Figure 1-11

INTERMEDIATE

6. A company pays an annual bonus to its employees. The bonus is calculated by multiplying the employee's annual salary by a bonus rate, which is based on the number of years the employee has been with the company. Employees working at the company for less than 5 years receive a 1% bonus; all others receive a 2% bonus. Using only the instructions shown in Figure 1-12, write two versions of an algorithm that prints each employee's bonus. Be sure to indent the instructions appropriately.

```
calculate the bonus by multiplying the salary by 1%
calculate the bonus by multiplying the salary by 2%
else
end if
end repeat
if (the years employed are greater than or equal to 5)
if (the years employed are less than 5)
print the bonus
enter the salary and years employed
repeat for each employee
```

Figure 1-12

INTERMEDIATE

7. Robin is standing in front of a flower bed that contains six flowers, as illustrated in Figure 1-13. Create an algorithm that directs Robin to pick the flowers as she walks to the other side of the flower bed. Robin should pick all white flowers with her right hand. Flowers that are not white should be picked with her left hand. Use only the instructions shown earlier in Figure 1-8.

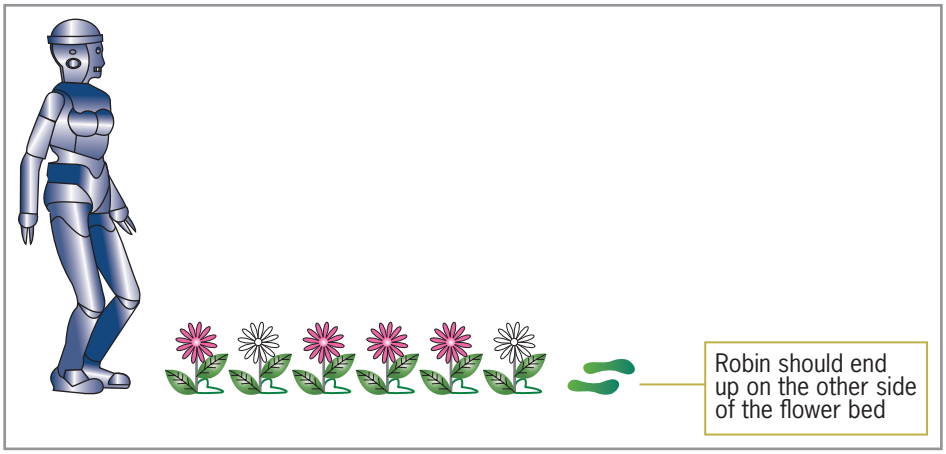


Figure 1-13

ADVANCED

8. The algorithm shown in Figure 1-14 should calculate and print the gross pay for five workers; however, some of the instructions are missing from the algorithm. Complete the algorithm. If an employee works more than 40 hours, he or she should receive time and one-half for the hours worked over 40.

```
_____
    enter the employee's name, hours worked, and pay rate
    _____
        calculate gross pay = hours worked times pay rate
    else
        calculate regular pay = pay rate times 40
        calculate overtime hours = hours worked minus 40
        calculate overtime pay = _____
        calculate gross pay = _____
    end if
    print the employee's name and gross pay
end repeat
```

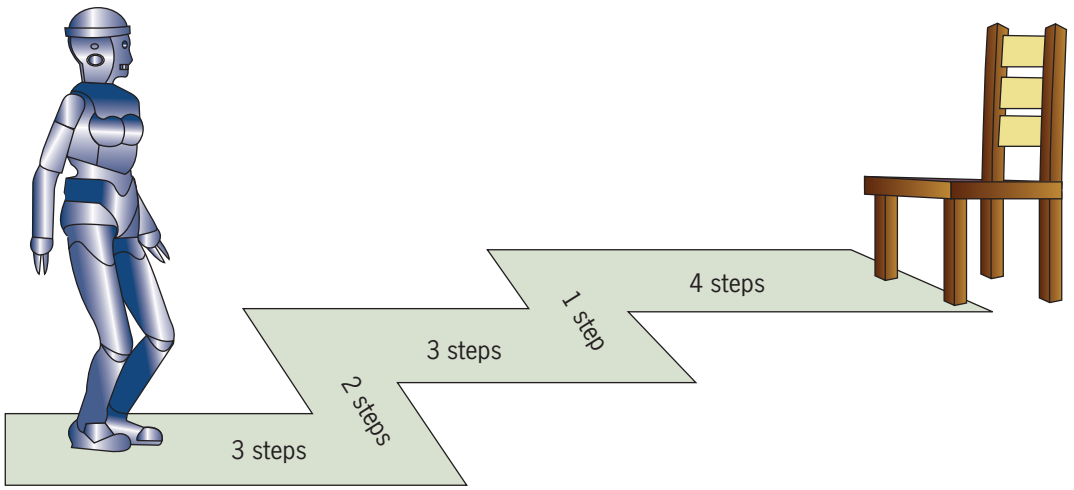
Figure 1-14

ADVANCED

9. Create an algorithm that tells someone how to evaluate the following expression: 12 / 2 + 3 * 2 – 3. The / operator means division, and the * operator means multiplication. (As you may remember from your math courses, division and multiplication are performed before addition and subtraction.)

10. The algorithm in Figure 1-15 should get Robin seated in the chair, but it does not work correctly. Correct the algorithm.

SWAT THE BUGS

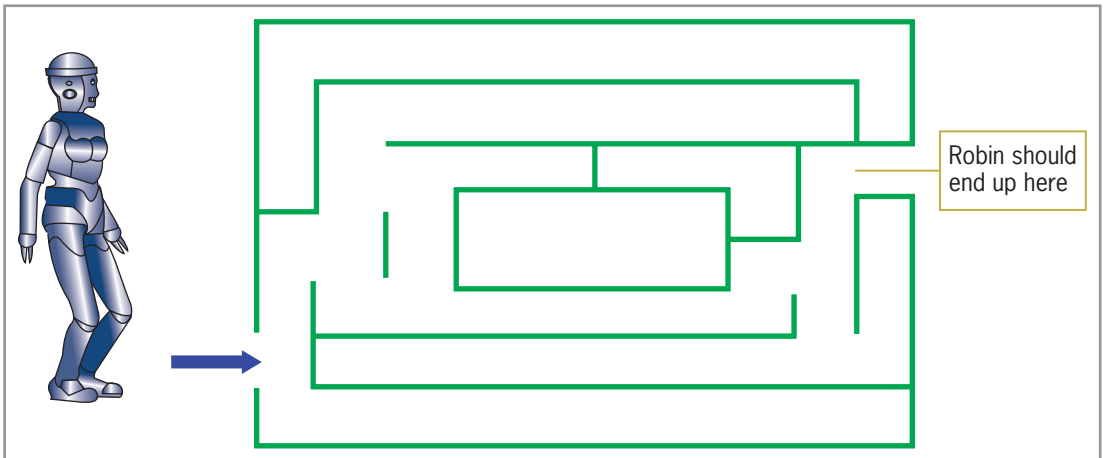



```
1. repeat 3 times
       walk forward
   end repeat
2. turn left 90 degrees
3. repeat 2 times
       walk forward
   end repeat
4. turn right 90 degrees
5. repeat 2 times
       walk forward
   end repeat
6. turn right 90 degrees
7. walk forward
8. turn right 90 degrees
9. repeat 4 times
       walk forward
   end repeat
10. turn right 180 degrees
11. sit down
```

Figure 1-15

SWAT THE BUGS

11. The algorithm in Figure 1-16 does not get Robin through the maze. Correct the algorithm.




```
1. Walk into the maze
2. turn left 90 degrees
3. repeat until you are directly in front of a wall
       walk forward
   end repeat
4. turn right 90 degrees
5. repeat until you are directly in front of a wall
       walk forward
   end repeat
6. turn right 90 degrees
7. repeat until you are directly i front of a wall
       walk forward
   end repeat
8. turn right 90 degrees
9. repeat until you are directly in front of a wall
       walk forward
   end repeat
10. turn right 90 degrees
11. repeat until you are directly in front of a wall
       walk forward
   end repeat
12. turn left 90 degrees
13. repeat until you are directly in front of a wall
       turn right 90 degrees
   end repeat
14. repeat until you are out of the maze
       walk forward
   end repeat
```

Figure 1-16

Answers to TRY THIS Exercises

1. See Figure 1-17.

```
1.  repeat 5 times
        walk forward
    end repeat
2.  jump over the box
3.  repeat until you are directly in front of the chair
        walk forward
    end repeat
4.  repeat 2 times
        turn right 90 degrees
    end repeat
5.  sit down
```

Figure 1-17

2. See Figure 1-18.

```
1.  enter the customer's age and the amount due
2.  if (the customer's age is greater than or equal to 65)
        calculate the amount due by multiplying the amount due by 90%
    end if
3.  print the amount due
```

Figure 1-18

CHAPTER 2

Beginning the Problem-Solving Process

After studying Chapter 2, you should be able to:

- Explain the problem-solving process used to create a computer program
- Analyze a problem
- Complete an IPO chart
- Plan an algorithm using pseudocode and flowcharts
- Desk-check an algorithm

Problem Solving

This chapter introduces you to the process that programmers follow when solving problems that require a computer solution. Although you may not realize it, you use a similar process to solve hundreds of small problems every day, such as how to get to school and what to do when you are hungry. Because most of these problems occur so often, you typically solve them almost automatically, without giving much thought to the process your brain goes through to arrive at the solutions. Unfortunately, problems that are either complex or unfamiliar usually cannot be solved so easily; most require extensive analysis and planning. Understanding the thought process involved in solving simple and familiar problems will make solving complex or unfamiliar ones easier. In this chapter, you will explore the thought process that you follow when solving common problems. You also will learn how to use a similar process to create a computer solution to a problem—in other words, to create a computer program. The computer solutions you create in this chapter will contain the sequence control structure only, in which each instruction is processed in order from beginning to end. Computer solutions requiring the selection structure are covered in Chapters 5 and 6, and those requiring the repetition structure are covered in Chapters 7 and 8.

Solving Everyday Problems

The first step in solving a familiar problem is to analyze the problem. Next, you plan, review, implement, evaluate, and modify (if necessary) the solution. Consider, for example, how you solve the problem of paying a bill that you received in the mail. First, your mind analyzes the problem to identify its important components. One very important component of any problem is the goal of solving the problem. In this case, the goal is to pay the bill. Other important components of a problem are the things that you can use to accomplish the goal. In this case, you will use the bill itself, as well as the preaddressed envelope that came with the bill. You also will use a bank check, pen, return address label, and postage stamp. After analyzing the problem, your mind plans an algorithm. Recall from Chapter 1 that an algorithm is the set of step-by-step instructions that describe how to accomplish a task. In other words, an algorithm is a solution to a problem. The current problem's algorithm, for example, describes how to use the bill, preaddressed envelope, bank check, pen, return address label, and postage stamp to pay the bill. Figure 2-1 shows a summary of the analysis and planning steps for the bill paying problem.

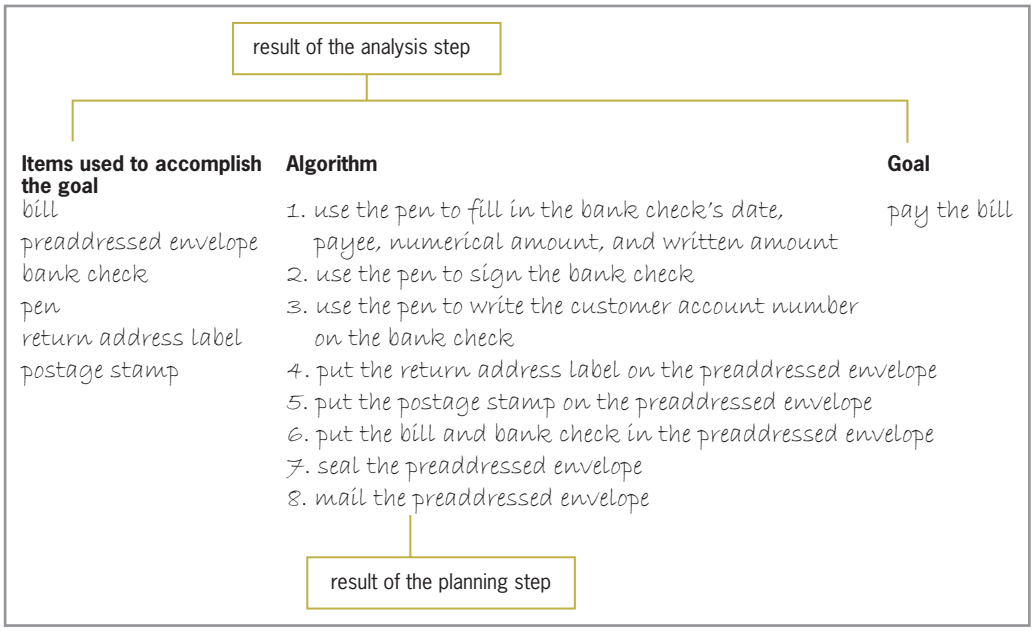

result of the analysis step
Items used to accomplish the goal
bill
preaddressed envelope
bank check
pen
return address label
postage stamp
Algorithm
1. use the pen to fill in the bank check's date, payee, numerical amount, and written amount
2. use the pen to sign the bank check
3. use the pen to write the customer account number on the bank check
4. put the return address label on the preaddressed envelope
5. put the postage stamp on the preaddressed envelope
6. put the bill and bank check in the preaddressed envelope
7. seal the preaddressed envelope
8. mail the preaddressed envelope
Goal
pay the bill
result of the planning step

Figure 2-1 Summary of the analysis and planning steps for the bill paying problem

After planning the algorithm, you review it (in your mind) to verify that it will work as intended. When you are satisfied that the algorithm is correct, you implement the algorithm by following each of its instructions in the order indicated. After implementing the algorithm, you evaluate it and, if necessary, you modify it. In this case, for example, you may decide to include the selection structure shown in instruction 6 in Figure 2-2.

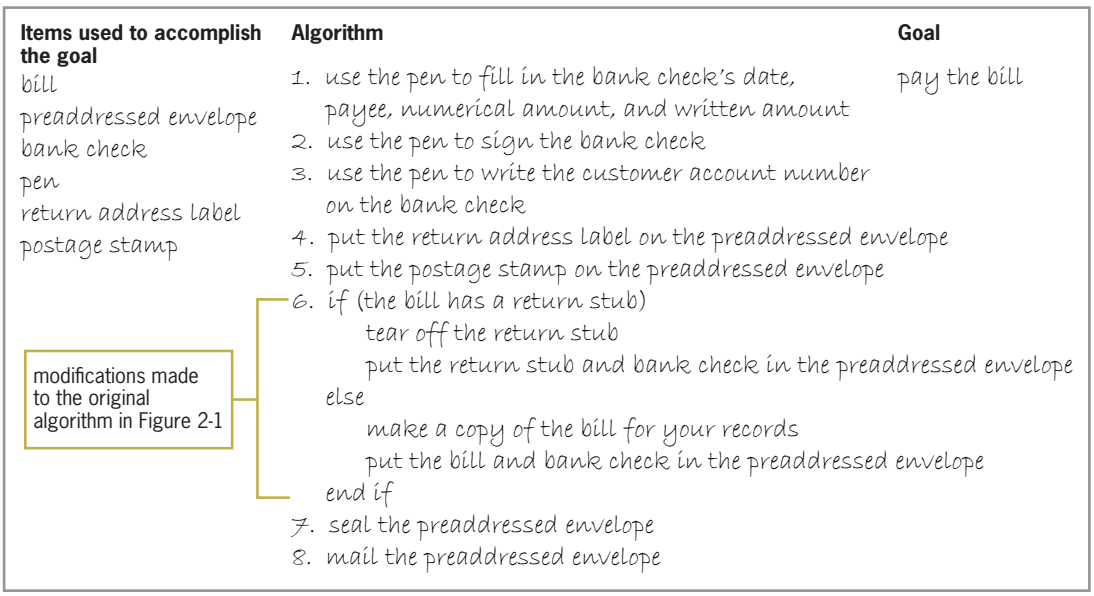

Items used to accomplish the goal
bill
preaddressed envelope
bank check
pen
return address label
postage stamp
Algorithm
1. use the pen to fill in the bank check's date, payee, numerical amount, and written amount
2. use the pen to sign the bank check
3. use the pen to write the customer account number on the bank check
4. put the return address label on the preaddressed envelope
5. put the postage stamp on the preaddressed envelope
6. if (the bill has a return stub)
tear off the return stub
put the return stub and bank check in the preaddressed envelope
else
make a copy of the bill for your records
put the bill and bank check in the preaddressed envelope
end if
7. seal the preaddressed envelope
8. mail the preaddressed envelope
Goal
pay the bill
modifications made to the original algorithm in Figure 2-1

Figure 2-2 Modified algorithm for the bill paying problem

Creating Computer Solutions to Problems

In the previous section, you learned how you create a solution to a familiar problem. A similar problem-solving process is used to create a computer program. A computer program also is a solution, but one that is implemented

with a computer. Figure 2-3 shows the steps that computer programmers follow when solving problems that require a computer solution.

This chapter covers the first three steps in the problem-solving process shown in Figure 2-3. The last three steps are covered in Chapters 3 and 4.

HOW TO Create a Computer Solution to a Problem

1. Analyze the problem
2. Plan the algorithm
3. Desk-check the algorithm
4. Code the algorithm into a program
5. Desk-check the program
6. Evaluate and modify (if necessary) the program

Figure 2-3 How to create a computer solution to a problem

Step 1—Analyze the Problem

You cannot solve a problem unless you understand it, and you cannot understand a problem unless you analyze it—in other words, unless you identify its important components. The two most important components of any problem are the problem's output and its input. The **output** is the goal of solving the problem, and the **input** is the item or items needed to achieve the goal. When analyzing a problem, you always search first for the output and then for the input. The first problem specification analyzed in this chapter is shown in Figure 2-4.

> Treyson Mobley wants a program that calculates and displays the amount he should tip a waiter at a restaurant. The program should subtract any liquor charge from the total bill and then calculate the tip (using a percentage) on the remainder.

Figure 2-4 Problem specification for Treyson Mobley

A helpful way to identify the output is to search the problem specification for an answer to the following question: *What does the user want to see displayed on the screen, printed on paper, or stored in a file?* The answer to this question typically is stated as nouns and adjectives in the problem specification. For instance, the problem specification in Figure 2-4 indicates that Treyson (the program's user) wants to see the amount of the waiter's tip displayed on the screen; therefore, the output is the tip. In this context, the word *tip* is a noun.

After determining the output, you then determine the input. A helpful way to identify the input is to search the problem specification for an answer to the following question: *What information will the computer need to know to display, print, or store the output items?* As with the output, the input typically is stated as nouns and adjectives in the problem specification. When determining the input, it helps to think about the information that you would need to solve the problem manually, because the computer will need to know the same information. In this case, to determine the tip, both you and the computer need to know the total bill, the liquor charge, and the tip percentage; these items, therefore, are the input. In this context, *total*, *liquor*, and *tip* are adjectives, while *bill*, *charge*, and *percentage* are nouns. This completes the analysis step for the Treyson Mobley problem. Some programmers use an **IPO chart** to organize and summarize the results of the analysis step, as

shown in Figure 2-5. **IPO** is an acronym for Input, Processing, and Output. You record the input items in the Input column of the IPO chart and record the output items in the Output column.

Input	Processing	Output
total bill liquor charge tip percentage	Processing items: Algorithm:	tip

Figure 2-5 Partially completed IPO chart showing the input and output items

Hints for Analyzing Problems

Unfortunately, analyzing real-world problems will not be as easy as analyzing the problems found in a textbook. The analysis step is the most difficult of the problem-solving steps, and it requires a lot of time, patience, and effort. If you are having trouble analyzing a problem, try reading the problem specification several times, as it is easy to miss information during the first reading. If the problem still is unclear to you, do not be shy about asking the user for more information. Remember, the more you understand a problem, the easier it will be for you to write a correct and efficient solution.

When reading a problem specification, it helps to use a pencil to lightly cross out the information that you feel is unimportant to the solution, as shown in Figure 2-6. Doing this reduces the amount of information you need to consider in your analysis. If you are not sure whether an item of information is important, ask yourself this question: *If I didn't know this information, could I still solve the problem?* If your answer is *Yes*, then the information is superfluous and you can ignore it. If you later find that the information is important, you can always erase the pencil line.

~~Treyson Mobley wants a program that~~ calculates and displays the amount he should tip ~~a waiter at a restaurant. The program should~~ subtract any liquor charge from the total bill and then calculate the tip (using a percentage) on the remainder.

Figure 2-6 Problem specification with unimportant information crossed out

Some problem specifications are difficult to analyze because they contain incomplete information. The problem specification shown in Figure 2-7 provides an example of this. It is clear from reading the problem specification that the output is the weekly gross pay, and the input is the hourly pay and the number of hours worked during the week. However, most companies pay a premium (such as time and one-half) for the hours worked over 40. You cannot tell whether the premium applies to the additional 10 hours that Jack worked, because the problem specification does not contain enough information. Before you can solve this problem, you will need to ask the payroll manager about the company's overtime policy.

Jack Osaki earns $7 per hour. Last week, Jack worked 50 hours. He wants a program that calculates and displays his weekly gross pay.

Figure 2-7 Problem specification that does not contain enough information

As a programmer, it is important to distinguish between information that truly is missing in the problem specification and information that simply is not stated, explicitly, in the problem specification—that is, information that is implied. For example, consider the problem specification shown in Figure 2-8. To solve the problem, you need to calculate the area of a rectangle; you do this by multiplying the rectangle's length by its width. Therefore, the area is the output, and the length and width are the input. Notice, however, that the words length and width do not appear in the problem specification. Although both items are not stated explicitly in the problem specification, neither is considered missing information. This is because the formula for calculating the area of a rectangle is common knowledge. (The formula also can be found in any math book or on the Internet.) With practice, you will be able to "fill in the gaps" in a problem specification also.

Caroline Casey wants a program that calculates and displays the area of any rectangle.

Figure 2-8 Problem specification in which the input is not explicitly stated

Mini-Quiz 2-1



The answers to Mini-Quiz questions are located in Appendix A.

Identify the output and input in each of the following problem specifications. Also identify any information that is missing from the specification.

1. Kendra Chopra lives in a state that charges a 5% sales tax. She wants a program that displays the amount of sales tax due on a purchase.

2. Henry Denton belongs to a CD (compact disc) club. Last year, he bought all of his CDs from the club at $8 per CD. He wants to know how much he saved last year by buying the CDs through the club rather than through a music store.

3. Kelsey Jones saves $1.50 per day. She would like to know the total amount she saved during the month of January.

4. If James Monet saves $2 per day, how much will he save in one year?

Step 2—Plan the Algorithm

The second step in the problem-solving process is to plan the algorithm that will transform the problem's input into its output. You record the algorithm in the Processing column of the IPO chart. Each instruction in the algorithm will describe an action that the computer needs to take. Therefore, each instruction should start with a verb. Most algorithms begin with an instruction to enter the input items into the computer. Next, you usually record instructions to process the input items to achieve the problem's output. The processing typically involves performing one or more calculations using the input items. Most algorithms end with an instruction to display, print, or

store the output items. *Display*, *print*, and *store* refer to the computer screen, the printer, and a file on a disk, respectively. Figure 2-9 shows the problem specification and IPO chart for the Treyson Mobley problem. The algorithm begins by entering the input items. It then uses the input items to calculate the output item. Notice that the algorithm states both *what* is to be calculated and *how* to calculate it. In this case, the tip is calculated by subtracting the liquor charge from the total bill and then multiplying the remainder by the tip percentage. The last instruction in the algorithm displays the output item. To avoid confusion, it is important that the algorithm is consistent when referring to the input and output items. For example, if the input item is listed as `total bill`, then the algorithm should refer to the item as `total bill` rather than a different name, such as `total` or `total due`.



Notice that each instruction in Figure 2-9's algorithm begins with a verb.

Problem specification
Treyson Mobley wants a program that calculates and displays the amount he should tip a waiter at a restaurant. The program should subtract any liquor charge from the total bill and then calculate the tip (using a percentage) on the remainder.

Input	Processing	Output
total bill liquor charge tip percentage	Processing items: none Algorithm: 1. enter the total bill, liquor charge, and tip percentage 2. calculate the tip by subtracting the liquor charge from the total bill and then multiplying the remainder by the tip percentage 3. display the tip	tip

Figure 2-9 Problem specification and IPO chart for the Treyson Mobley problem

The algorithm in Figure 2-9 is composed of short English statements, referred to as **pseudocode**. The word *pseudocode* means *false code*. It's called false code because, although it resembles programming language instructions, pseudocode cannot be understood by a computer. Programmers use pseudocode to help them while they are planning an algorithm. It allows them to jot down their ideas using a human-readable language without having to worry about the syntax of the programming language itself. The pseudocode is used as a guide when the programmer codes the algorithm. Coding the algorithm is the fourth step in the problem-solving process and is covered in Chapters 3 and 4. Pseudocode is not standardized; every programmer has his or her own version, but you will find some similarities among the various versions. Although the word *pseudocode* might be unfamiliar to you, you already have written pseudocode without even realizing it. Think about the last time you gave directions to someone. You wrote down each direction on paper, in your own words. Your directions were a form of pseudocode.

Besides using pseudocode, programmers also use flowcharts when planning algorithms. Unlike pseudocode, a **flowchart** uses standardized symbols to visually depict an algorithm. You can draw the flowchart symbols by hand; or, you can use the drawing or shapes feature in a word processor. You also can use a flowcharting program, such as SmartDraw or Visio. Figure 2-10 shows the algorithm from Figure 2-9 in flowchart form. The flowchart contains three different symbols: an oval, a parallelogram, and a rectangle. The symbols are

connected with lines, called **flowlines**. The oval symbol is called the **start/stop symbol**. The start oval indicates the beginning of the flowchart, and the stop oval indicates the end of the flowchart. Between the start and stop ovals are two parallelograms, called input/output symbols. You use the **input/output symbol** to represent input tasks (such as getting information from the user) and output tasks (such as displaying, printing, or storing information). The first parallelogram in Figure 2-10 represents an input task, while the last parallelogram represents an output task. The rectangle in a flowchart is called the **process symbol** and is used to represent tasks such as calculations.

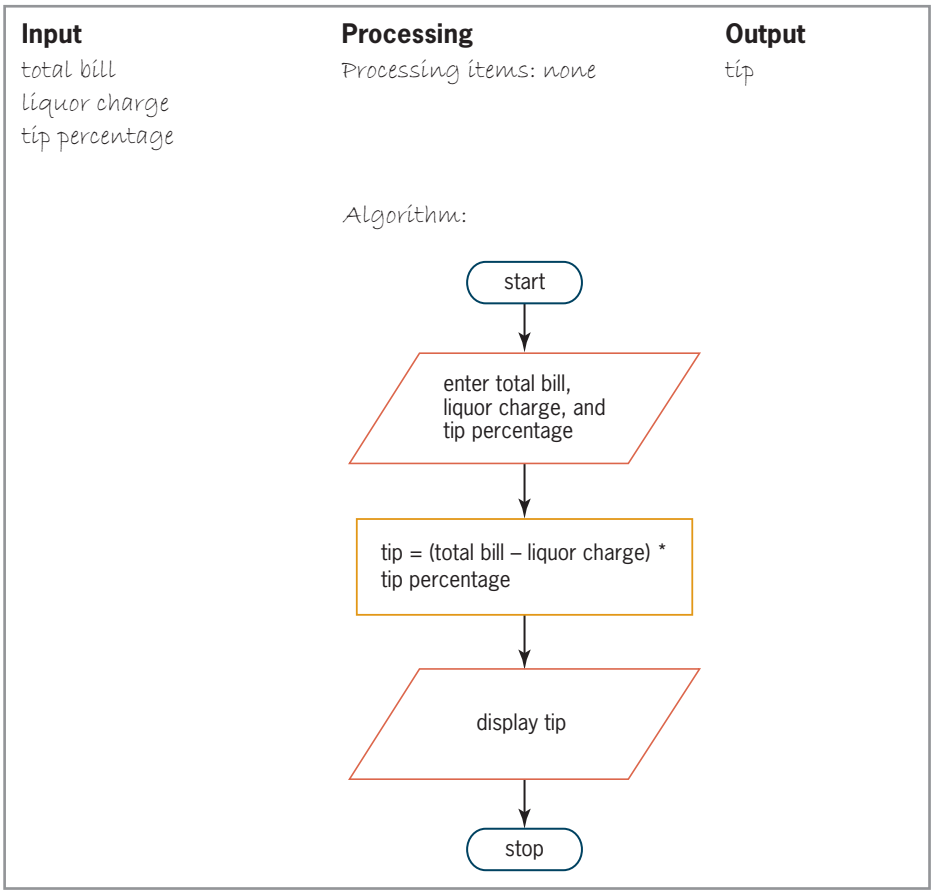


Figure 2-10 Figure 2-9's algorithm in flowchart form

When planning an algorithm, you do not need to create both pseudocode and a flowchart; you need to use only one of these planning tools. The tool you use is really a matter of personal preference. For simple algorithms, pseudocode works just fine. However, when an algorithm becomes more complex, its logic may be easier to see in a flowchart. As the old adage goes, a picture is sometimes worth a thousand words.

Even a very simple problem can have more than one solution. Figure 2-11 shows a different solution to the Treyson Mobley problem. In this solution, the difference between the total bill and liquor charge is calculated in a separate instruction rather than in the instruction that calculates the tip. The total bill without liquor charge item is neither an input item (because it's not provided by the user) nor an output item (because it won't be displayed, printed, or stored in a file). Instead, the total bill without liquor charge is a special item, commonly referred to as a processing item. A **processing item**

represents an intermediate value (neither input nor output) that the algorithm uses when processing the input into the output. In this case, the algorithm uses two of the input items (total bill and liquor charge) to calculate the total bill without liquor charge (an intermediate value). It then uses this intermediate value, along with the tip percentage, to compute the tip.

Problem specification
Treyson Mobley wants a program that calculates and displays the amount he should tip a waiter at a restaurant. The program should subtract any liquor charge from the total bill and then calculate the tip (using a percentage) on the remainder.

Input	Processing	Output
total bill liquor charge tip percentage	Processing items: total bill without liquor charge	tip

Algorithm (pseudocode):
1. enter the total bill, liquor charge, and tip percentage
2. calculate the total bill without liquor charge by subtracting the liquor charge from the total bill
3. calculate the tip by multiplying the total bill without liquor charge by the tip percentage
4. display the tip

Algorithm (flowchart):

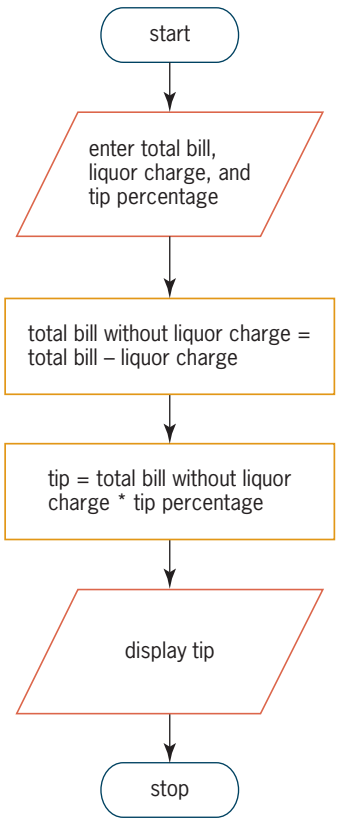

start
enter total bill, liquor charge, and tip percentage
total bill without liquor charge = total bill – liquor charge
tip = total bill without liquor charge * tip percentage
display tip
stop

Figure 2-11 A different solution to the Treyson Mobley problem

The algorithms shown in Figures 2-9 through 2-11 produce the same result and simply represent two different ways of solving the same problem.

Mini-Quiz 2-2



The answers to Mini-Quiz questions are located in Appendix A.

1. The parallelogram in a flowchart is called the _____ symbol.

2. In a flowchart, calculation tasks are placed in a processing symbol, which has a _____ shape.

3. Kendra Chopra lives in a state that charges a 5% sales tax. She wants a program that displays the amount of sales tax due on a purchase. The output is the sales tax. The input is the purchase amount and the sales tax rate. Complete an appropriate IPO chart, using pseudocode in the Algorithm section.

4. Henry Denton belongs to a CD (compact disc) club. Last year, he bought all of his CDs from the club at $8 per CD. He wants to know how much he saved last year by buying the CDs through the club rather than through a music store that charges $12 per CD. The output is the savings. The input is the number of CDs purchased, the club CD price, and the store CD price. Complete an appropriate IPO chart, using a flowchart in the Algorithm section. The algorithm should use two processing items: one for the cost of buying the CDs through the club and the other for the cost of buying the CDs through the store.

Step 3—Desk-Check the Algorithm

After analyzing a problem and planning its algorithm, you then desk-check the algorithm. The term **desk-checking** refers to the fact that the programmer reviews the algorithm while seated at his or her desk rather than in front of the computer. Desk-checking is also called **hand-tracing**, because the programmer uses a pencil and paper to follow each of the algorithm's instructions by hand. You desk-check an algorithm to verify that it is not missing any instructions and that the existing instructions are correct and in the proper order. Before you begin the desk-check, you first choose a set of sample data for the input values, which you then use to manually compute the expected output value. For the Treyson Mobley solution, you will use input values of $45, $10, and .2 (the decimal equivalent of 20%) as the total bill, liquor charge, and tip percentage, respectively. A manual calculation of the tip results in $7, as shown in Figure 2-12.

```
$ 45   (total bill)
-  10  (liquor charge)
   35  (total bill without liquor charge)
*  .2  (tip percentage)
$   7  (tip)
```

Figure 2-12 Manual tip calculation for the first desk-check

You now use the sample input values to desk-check the algorithm, which should result in the expected output value of $7. It is helpful to use a desk-check

table when desk-checking an algorithm. The table should contain one column for each input item listed in the IPO chart, as well as one column for each output item and one column for each processing item (if any). You can perform the desk-check using either the algorithm's pseudocode or its flowchart. Figure 2-13 shows one solution for the Treyson Mobley problem along with a partially completed desk-check table. (The flowchart for this solution is shown earlier in Figure 2-11.) Notice that the desk-check table contains five columns: three for the input items, one for the processing item, and one for the output item.

Input	Processing	Output
total bill liquor charge tip percentage	Processing items: total bill without liquor charge Algorithm: 1. enter the total bill, liquor charge, and tip percentage 2. calculate the total bill without liquor charge by subtracting the liquor charge from the total bill 3. calculate the tip by multiplying the total bill without liquor charge by the tip percentage 4. display the tip	tip

total bill	liquor charge	tip percentage	total bill without liquor charge	tip

Figure 2-13 Treyson Mobley solution and partially completed desk-check table

The first instruction in the algorithm is to enter the input values. You record the results of this instruction by writing 45, 10, and .2 in the total bill, liquor charge, and tip percentage columns, respectively, in the desk-check table. See Figure 2-14.

total bill	liquor charge	tip percentage	total bill without liquor charge	tip
45	10	.2		

Figure 2-14 Input values entered in the desk-check table

The second instruction in the algorithm is to calculate the total bill without liquor charge by subtracting the liquor charge from the total bill. The desk-check table shows that the total bill is 45 and the liquor charge is 10. When making the calculation, always use the table to determine the values of the total bill and liquor charge. Doing this helps to verify the accuracy of the algorithm. If, for example, the table did not show any amount in the total bill column, you would know that your algorithm missed an instruction; in this case, it neglected to enter the total bill amount. When you subtract the liquor charge (10) from the total bill (45), you get 35. You record the number 35 in the total bill without liquor charge column, as shown in Figure 2-15.

total bill	liquor charge	tip percentage	total bill without liquor charge	tip
45	10	.2	35	

Figure 2-15 Processing item's value entered in the desk-check table

The third instruction in the algorithm is to calculate the tip by multiplying the total bill without liquor charge by the tip percentage. The desk-check table shows that the total bill without liquor charge is 35 and the tip percentage is .2. When you multiply 35 by .2, you get 7. You record the number 7 in the tip column, as shown in Figure 2-16.

total bill	liquor charge	tip percentage	total bill without liquor charge	tip
45	10	.2	35	7

Figure 2-16 Output value entered in the desk-check table

The last instruction in the algorithm is to display the tip. In this case, the number 7 will be displayed because that is what appears in the tip column. Notice that this amount agrees with the manual calculation shown in Figure 2-12; therefore, the algorithm appears to be correct. The only way to know for sure, however, is to test the algorithm a few more times with different input values. For the second desk-check, you will test the algorithm using $30, $0, and .15 as the total bill, liquor charge, and tip percentage, respectively. The tip should be $4.50, as shown in Figure 2-17.

```
$   30   (total bill)
-    0   (liquor charge)
    ---
    30   (total bill without liquor charge)
*  .15   (tip percentage)
    ---
$ 4.50   (tip)
```

Figure 2-17 Manual tip calculation for the second desk-check

Recall that the first instruction in the algorithm is to enter the total bill, liquor charge, and tip percentage. Therefore, you write 30, 0, and .15 in the appropriate columns in the desk-check table, as shown in Figure 2-18. Although it's not required, some programmers find it helpful to lightly cross out the previous value in a column before recording a new value. Doing this helps keep track of the column's current value.

total bill	liquor charge	tip percentage	total bill without liquor charge	tip
~~45~~	~~10~~	~~.2~~	35	~~7~~
30	0	.15		

Figure 2-18 Second set of input values entered in the desk-check table

The second instruction in the algorithm is to calculate the total bill without liquor charge. You do this by subtracting the value in the liquor charge column (0) from the value in the total bill column (30). You record the result (30) in the total bill without liquor charge column. See Figure 2-19.

total bill	liquor charge	tip percentage	total bill without liquor charge	tip
~~45~~	~~10~~	~~.2~~	~~35~~	~~7~~
30	0	.15	30	

Figure 2-19 Value of the second desk-check's processing item entered in the desk-check table

The third instruction in the algorithm is to calculate the tip by multiplying the value in the total bill without liquor charge column (30) by the value in the tip percentage column (.15). You record the result (4.50) in the tip column, as shown in Figure 2-20. The last instruction in the algorithm is to display the tip. In this case, the number 4.50 will be displayed, which agrees with the manual calculation shown in Figure 2-17.

total bill	liquor charge	tip percentage	total bill without liquor charge	tip
~~45~~	~~10~~	~~.2~~	~~35~~	~~7~~
30	0	.15	30	4.50

Figure 2-20 Value of the second desk-check's output item entered in the desk-check table

To be sure an algorithm works correctly, you should desk-check it several times using both valid and invalid data. **Valid data** is data that the algorithm is expecting the user to enter. For example, the algorithm that you just finished desk-checking expects the user to provide positive numbers for the input values. **Invalid data** is data that the algorithm is not expecting the user to enter, such as a negative number for the total bill. You should test an algorithm with invalid data because users sometimes make mistakes when entering data. In later chapters in this book, you will learn how to write algorithms that correctly handle input errors. For now, however, you can assume that the user will always enter valid data.

The Gas Mileage Problem

The gas mileage problem will help reinforce what you learned in this chapter. Figure 2-21 shows the problem specification.

> When Cheryl Harrison began her trip from New York to Wyoming, she filled her car's tank with gas and reset its trip meter to zero. After traveling 324 miles, Cheryl stopped at a gas station to refuel; the gas tank required 17 gallons. Cheryl wants a program that calculates and displays her car's gas mileage at any time during the trip. The gas mileage is the number of miles her car was driven per gallon of gas.

Figure 2-21 Problem specification for the gas mileage problem

First, analyze the problem, looking for nouns and adjectives that represent both the output and the input. The output should answer the following question: *What does the user want to see displayed on the screen, printed on paper, or stored in a file?* The input should answer the question: *What information will the computer need to know to display, print, or store the output items?* In the gas mileage problem, the output is the miles per gallon and the input is the miles driven and gallons used.

Next, plan the algorithm. Recall that most algorithms begin with an instruction to enter the input items into the computer, followed by instructions that process the input items and then display, print, or store the output items. Figure 2-22 shows the completed IPO chart for the gas mileage problem.

Input	Processing	Output
miles driven gallons used	Processing items: none Algorithm: 1. enter the miles driven and gallons used 2. calculate the miles per gallon by dividing the miles driven by the gallons used 3. display the miles per gallon	miles per gallon

each instruction begins with a verb

Figure 2-22 IPO chart for the gas mileage problem

After planning the algorithm, you then desk-check it. You will desk-check the algorithm twice, first using 324 and 17 as the miles driven and gallons used, respectively, and then using 200 and 12. Figure 2-23 shows the completed desk-check table for the gas mileage problem. (The miles per gallon are rounded to two decimal places.)

miles driven	gallons used	miles per gallon
~~324~~	~~17~~	~~19.06~~
200	12	16.67

Figure 2-23 Desk-check table for the gas mileage problem

Mini-Quiz 2-3



The answers to Mini-Quiz questions are located in Appendix A.

1. Desk-check the algorithm shown in Figure 2-24 twice. First, use a purchase amount of $67 and a sales tax rate of .05 (the decimal equivalent of 5%). Then use a purchase amount of $100 and a sales tax rate of .02 (the decimal equivalent of 2%).

2. Desk-check the algorithm shown in Figure 2-25 twice. Use the numbers 5 and 11 as the first set of input values. Use the numbers 6 and 12 as the second set of input values.

Input	Processing	Output
purchase amount sales tax rate	Processing items: none Algorithm: 1. enter the purchase amount and sales tax rate 2. calculate the sales tax by multiplying the purchase amount by the sales tax rate 3. display the sales tax	sales tax

Figure 2-24 IPO chart for Question 1 in Mini-Quiz 2-3

Input	Processing	Output
first number second number	Processing items: sum Algorithm: 1. enter the first number and second number 2. calculate the sum by adding together the first number and second number 3. calculate the average by dividing the sum by 2 4. display the average	average

Figure 2-25 IPO chart for Question 2 in Mini-Quiz 2-3



The answers to the labs are located in Appendix A.



LAB 2-1 Stop and Analyze

Aiden Nelinski is paid every Friday. He is scheduled to receive either a 2% or 2.5% raise next week. He wants a program that calculates and displays the amount of his new weekly pay. Study the IPO chart and desk-check table shown in Figures 2-26 and 2-27; and then answer the questions.

Input	Processing	Output
current weekly pay pay raise percentage	Processing items: none Algorithm: 1. enter the current weekly pay and raise percentage 2. calculate the new weekly pay by multiplying the current weekly pay by the raise percentage and then adding the result to the current weekly pay 3. display the new weekly pay	new weekly

Figure 2-26 IPO chart for Lab 2-1

current weekly pay	raise percentage	new weekly pay
~~300~~	~~.02~~	~~306~~
500	.025	512.50

Figure 2-27 Desk-check table for Lab 2-1

QUESTIONS

1. What will the algorithm in Figure 2-26 display when the user enters 300 and .02 as the current weekly pay and raise percentage, respectively? What will it display when the user enters 500 and .025?
2. How would you modify Lab 2-1's IPO chart and desk-check table to include the amount of the raise as a processing item?
3. How would you modify Lab 2-1's IPO chart and desk-check table to also display the amount of the raise?



LAB 2-2 Plan and Create

In this lab, you will plan and create an algorithm for the manager of the Lakeview Hotel. The problem specification is shown in Figure 2-28.

> The manager of the Lakeview Hotel wants a program that calculates and displays a guest's total bill. Each guest pays a room charge that is based on a per-night rate. For example, if the per-night rate is $100 and the guest stays two nights, the room charge is $200. Customers also may incur a one-time room service charge and a one-time telephone charge.

Figure 2-28 Problem specification for Lab 2-2

First, analyze the problem, looking for the output first and then for the input. Recall that the output and input typically are stated as nouns and adjectives in the problem specification. Asking the question *What does the user want to see displayed on the screen, printed on paper, or stored in a file?* will help you determine the output. In this case, the manager wants to see the guest's total bill displayed on the computer screen. The question *What information will the computer need to know to display, print, or store the output items?* will help you determine the input. In this case, the input is the number of nights, per-night rate, room service charge, and telephone charge. Figure 2-29 shows the input and output items entered in an IPO chart.

Input	Processing	Output
number of nights per-night rate room service charge telephone charge	Processing items: Algorithm:	total bill

Figure 2-29 Partially completed IPO chart for Lab 2-2

After determining a problem's output and input, you then plan its algorithm. Recall that most algorithms begin by entering the input items into the computer. The first instruction in the current problem's algorithm, for example, will be enter the number of nights, per-night rate, room service charge,

and telephone charge. Notice that the instruction refers to the input items using the same names listed in the Input column of the IPO chart. After the instruction to enter the input items, you usually record instructions to process those items, typically including the items in one or more calculations. In this case, you first will use the number of nights and per-night rate to calculate an intermediate value: the room charge. You then will calculate the total bill by adding together the room charge, room service charge, and telephone charge. Recall that most algorithms end with an instruction to display, print, or store the output items. The last instruction in this algorithm will simply display the total bill on the screen. Figure 2-30 shows the completed IPO chart. Notice that each instruction in the algorithm begins with a verb.

Input	Processing	Output
number of nights per-night rate room service charge telephone charge	Processing items: room charge Algorithm: 1. enter the number of nights, per-night rate, room service charge, and telephone charge 2. calculate the room charge by multiplying the number of nights by the per-night rate 3. calculate the total bill by adding together the room charge, room service charge, and telephone charge 4. display the total bill	total bill

Figure 2-30 Completed IPO chart for Lab 2-2

You also can write the algorithm in Figure 2-30 without using a processing item, as shown in Figure 2-35.

After completing the IPO chart, you then move on to the third step in the problem-solving process, which is to desk-check the algorithm. You begin by choosing a set of sample data for the input values. You then use the values to manually compute the expected output. You will desk-check the current algorithm twice: first using 4, $100, $12, and $5 as the number of nights, per-night rate, room service charge, and telephone charge, respectively, and then using 2, $55, $20, and $0. For the first desk-check, the total bill should be $417. For the second desk-check, the total bill should be $130. The manual calculations for both desk-checks are shown in Figure 2-31.

First desk-check		Second desk-check	
4	(number of nights)	2	(number of nights)
* 100	(per-night rate)	* 55	(per-night rate)
400	(room charge)	110	(room charge)
+ 12	(room service charge)	+ 20	(room service charge)
+ 5	(telephone charge)	+ 0	(telephone charge)
$ 417	(total bill)	$ 130	(total bill)

Figure 2-31 Manual total bill calculations for the two desk-checks

Next, you create a desk-check table that contains one column for each input, processing, and output item. You then begin desk-checking the algorithm. The first instruction is to enter the input values. Figure 2-32 shows these values entered in the desk-check table.

number of nights	per-night rate	room service charge	telephone charge	room charge	total bill
4	100	12	5		

Figure 2-32 First set of input values entered in the desk-check table

The second and third instructions are to calculate the room charge and total bill. Figure 2-33 shows these values entered in the desk-check table.

number of nights	per-night rate	room service charge	telephone charge	room charge	total bill
4	100	12	5	400	417

Figure 2-33 Room charge and total bill values entered in the desk-check table

The last instruction in the algorithm is to display the total bill. According to the desk-check table in Figure 2-33, the total bill is 417. This amount agrees with the manual calculation shown earlier in Figure 2-31. Now use the second set of input values to desk-check the algorithm. Figure 2-34 shows the result of the second desk-check. Notice that the amount in the total bill column (130) agrees with the manual calculation shown earlier in Figure 2-31.

number of nights	per-night rate	room service charge	telephone charge	room charge	total bill
~~4~~	~~100~~	~~12~~	~~5~~	~~400~~	~~417~~
2	55	20	0	110	130

Figure 2-34 Desk-check table showing the result of the second desk-check

Almost every problem, even simple ones, can be solved in more than one way. Figure 2-35 shows another solution to Lab 2-2's problem. Unlike the solution shown in Figure 2-30, this solution does not use a processing item.

Problem specification
The manager of the Lakeview Hotel wants a program that calculates and displays a guest's total bill. Each guest pays a room charge that is based on a per-night rate. For example, if the per-night rate is $100 and the guest stays two nights, the room charge is $200. Customers also may incur a one-time room service charge and a one-time telephone charge.

Input	Processing	Output
number of nights per-night rate room service charge telephone charge	Processing items: none Algorithm: 1. enter the number of nights, per-night rate, room service charge, and telephone charge 2. calculate the total bill by multiplying the number of nights by the per-night rate and then adding the result to the room service charge and telephone charge 3. display the total bill	total bill

Figure 2-35 A different solution for Lab 2-2's problem



LAB 2-3 Modify

Each guest at the Lakeview Hotel pays an entertainment tax, which is a percentage of the room charge only. Make the appropriate modifications to the IPO chart shown in Figure 2-30. Desk-check the algorithm twice. For the first desk-check, use 3, $70, $0, $10, and .05 as the number of nights, per-night rate, room service charge, telephone charge, and entertainment tax rate, respectively. For the second desk-check, use 7, $100, $25, $6, and .03.



LAB 2-4 Desk-Check

The algorithm shown in Figure 2-36 calculates and displays an annual property tax. Currently, the property tax rate is $1.50 for each $100 of a property's assessed value; however, the tax rate changes each year. Desk-check the algorithm three times. For the first desk-check, use $104,000 and $1.50 as the assessed value and property tax rate, respectively. For the second desk-check, use $239,000 and $1.15. For the third desk-check, use $86,000 and $0.98. Be sure to manually calculate the annual property tax for each set of input values. (The annual property tax using the first set of input values is $1560.)

Input	Processing	Output
assessed value tax rate	Processing items: none Algorithm: 1. enter the assessed value and tax rate 2. calculate the annual property tax by dividing the assessed value by 100 and then multiplying the result by the tax rate 3. display the annual property tax	annual property tax

Figure 2-36 IPO chart for Lab 2-4



LAB 2-5 Debug

The algorithm in Figure 2-37 should calculate and display the average of three numbers, but it is not working correctly. In this lab, you will find and correct the errors in the algorithm.

Input	Processing	Output
first number second number third number	Processing items: sum Algorithm: 1. enter the first number, second number, and third number 2. calculate the average by dividing the sum by 3 3. display the average number	average

Figure 2-37 IPO chart for Lab 2-5

You locate the errors in an algorithm by desk-checking it. First, choose a set of sample data for the input values. In this case, you will use the numbers 25, 63, and 14. Now use the values to manually compute the expected output—in this case, the average. The average of the three numbers is 34. Next, create a desk-check table that contains a column for each input, processing, and output item. This desk-check table will contain five columns. Finally, walk through each of the instructions in the algorithm, recording the appropriate values in the desk-check table. The first instruction in the algorithm in Figure 2-37 is to enter the three input values. Figure 2-38 shows these values entered in the desk-check table.

first number	second number	third number	sum	average
25	63	14		

Figure 2-38 Three input values entered in the desk-check table

The next instruction is to calculate the average by dividing the sum by 3. Notice that the sum column in the desk-check table does not contain a value.

This fact alerts you that the algorithm is missing an instruction. In this case, it is missing the instruction to calculate the sum of the three numbers. The missing instruction is shaded in Figure 2-39.

Input	Processing	Output
first number second number third number	Processing items: sum Algorithm: 1. enter the first number, second number, and third number 2. calculate the sum by adding together the first number, second number, and third number 3. calculate the average by dividing the sum by 3 4. display the average number	average

Figure 2-39 Missing instruction added to the IPO chart for Lab 2-5

The additional instruction calculates the sum by adding together the first number, second number, and third number. According to the desk-check table shown earlier in Figure 2-38, those values are 25, 63, and 14. The sum of those values is 102. Figure 2-40 shows the sum entered in the desk-check table.

first number	second number	third number	sum	average
25	63	14	102	

Figure 2-40 Sum entered in the desk-check table

The next instruction is to calculate the average by dividing the sum by 3. According to the desk-check table, the sum is 102. Dividing 102 by 3 results in 34. Figure 2-41 shows the average entered in the desk-check table.

first number	second number	third number	sum	average
25	63	14	102	34

Figure 2-41 Average entered in the desk-check table

The last instruction in the algorithm is *display the average number*. Notice that the desk-check table does not contain a column with the heading "average number." Recall that, to avoid confusion, it is important to be consistent when referring to the input, output, and processing items in the IPO chart. In this case, the last instruction in the algorithm should be *display the average* rather than *display the average number*. According to the desk-check table, the average column contains the number 34, which is correct. Figure 2-42 shows the corrected algorithm. The changes made to the original algorithm (shown earlier in Figure 2-37) are shaded in the figure.

Input	Processing	Output
first number second number third number	Processing items: sum Algorithm: 1. enter the first number, second number, and third number 2. calculate the sum by adding together the first number, second number, and third number 3. calculate the average by dividing the sum by 3 4. display the average	average

Figure 2-42 Corrected algorithm for Lab 2-5

On your own, desk-check the corrected algorithm shown in Figure 2-42 using the numbers 33, 56, and 70.

Summary

- The process you follow when creating solutions to everyday problems is similar to the process used to create a computer program, which also is a solution to a problem. This problem-solving process typically involves analyzing the problem and then planning, reviewing, implementing, evaluating, and modifying (if necessary) the solution.
- Programmers use tools such as IPO (Input, Processing, Output) charts, pseudocode, and flowcharts to help them analyze problems and develop algorithms.
- The first step in the problem-solving process is to analyze the problem. During the analysis step, the programmer first determines the output, which is the goal or purpose of solving the problem. The programmer then determines the input, which is the information needed to reach the goal.
- The second step in the problem-solving process is to plan the algorithm. During the planning step, programmers write the instructions that will transform the input into the output. Most algorithms begin by entering some data (the input items), then processing that data (usually by performing some calculations), and then displaying some data (the output items).
- The third step in the problem-solving process is to desk-check the algorithm to determine whether it will work as intended. First, choose a set of sample data for the input values. Then use the values to manually compute the expected output. Next, create a desk-check table that contains a column for each input, processing, and output item. Finally, walk through each of the instructions in the algorithm, recording the appropriate values in the desk-check table.

Key Terms

Desk-checking—the process of manually walking through each of the instructions in an algorithm; also called hand-tracing

Flowchart—a tool that programmers use to help them plan (or depict) an algorithm; consists of standardized symbols connected by flowlines

Flowlines—the lines that connect the symbols in a flowchart

Hand-tracing—another term for desk-checking

Input—the items a program needs in order to achieve the output

Input/output symbol—the parallelogram in a flowchart; used to represent input and output tasks

Invalid data—data that the algorithm is not expecting the user to enter

IPO—an acronym for Input, Processing, and Output

IPO chart—a chart that some programmers use to organize and summarize the results of a problem analysis

Output—the goal of solving a problem; the items the user wants to display, print, or store

Process symbol—the rectangle symbol in a flowchart; used to represent tasks such as calculations

Processing item—an intermediate value (neither input nor output) that an algorithm uses when processing the input into the output

Pseudocode—a tool that programmers use to help them plan an algorithm; consists of short English statements; means *false code*

Start/stop symbol—the oval symbol in a flowchart; used to mark the beginning and end of the flowchart

Valid data—data that the algorithm is expecting the user to enter

Review Questions

1. The first step in the problem-solving process is to _____.
   a. plan the algorithm
   b. analyze the problem
   c. desk-check the algorithm
   d. code the algorithm

2. Programmers refer to the goal of solving a problem as the _____.
   a. input
   b. output
   c. processing
   d. purpose

3. Programmers refer to the items needed to reach a problem's goal as the _____.

   a. input

   b. output

   c. processing

   d. purpose

4. A problem's _____ will answer the question *What does the user want to see displayed on the screen, printed on the printer, or stored in a file?*

   a. input

   b. output

   c. processing

   d. purpose

5. A problem's _____ will answer the question *What information will the computer need to know to display, print, or store the output items?*

   a. input

   b. output

   c. processing

   d. purpose

6. The calculation instructions in an algorithm should state _____.

   a. only *what* is to be calculated

   b. only *how* to calculate something

   c. both *what* is to be calculated and *how* to calculate it

   d. both *what* is to be calculated and *why* it is calculated

7. Most algorithms follow the format of _____.

   a. entering the input items, then displaying, printing, or storing the input items, and then processing the output items

   b. entering the input items, then processing the output items, and then displaying, printing, or storing the output items

   c. entering the input items, then processing the input items, and then displaying, printing, or storing the output items

   d. entering the output items, then processing the output items, and then displaying, printing, or storing the output items

8. The short English statements that represent an algorithm are called _____.
   a. flow diagrams
   b. IPO charts
   c. pseudocharts
   d. pseudocode

9. The oval in a flowchart is called the _____ symbol.
   a. calculation
   b. input/output
   c. process
   d. start/stop

10. A desk-check table should contain _____.
    a. one column for each input item
    b. one column for each output item
    c. one column for each processing item
    d. all of the above

Exercises



Pencil and Paper

TRY THIS

1. The sales manager at Colfax Products wants a program that allows him to enter the sales made in each of two states. The program should calculate and display the commission, which is a percentage of the total sales. Complete an IPO chart for this problem. Plan the algorithm using a flowchart. Also complete a desk-check table for your algorithm. For the first desk-check, use $1000 and $2000 as the two state sales and use .05 (the decimal equivalent of 5%) as the commission rate. Then use $3000 and $2500 as the two state sales, and use .06 as the commission rate. (The answers to TRY THIS Exercises are located at the end of the chapter.)

TRY THIS

2. Party-On sells individual hot/cold cups and dessert plates for parties. Sue Chen wants a program that allows her to enter the price of a cup, the price of a plate, the number of cups purchased, the number of plates purchased, and the sales tax rate. The program should calculate and display the total cost of the purchase. Complete an IPO chart for this problem. Plan the algorithm using pseudocode. Desk-check the algorithm using $.50 as the cup price, $1 as the plate price, 35 as the number of cups, 35 as the number of plates, and .02 as the sales tax rate. Then desk-check it using $.25, $.75, 20, 10, and .06. (The answers to TRY THIS Exercises are located at the end of the chapter.)

3. Modify the IPO chart shown earlier in Figure 2-9 as follows. Treyson will be charging the total bill, including the tip, to his credit card. Modify the solution so that, in addition to calculating and displaying the appropriate tip, it also calculates and displays the amount charged to Treyson's credit card. Desk-check the algorithm using $50 as the total bill, $5 as the liquor charge, and .2 as the tip percentage. Then desk-check it using $15 as the total bill, $0 as the liquor charge, and .15 as the tip.

MODIFY THIS

4. Wilma Peterson is paid by the hour. She wants a program that calculates her weekly gross pay. For this exercise, you do not need to worry about overtime pay, as Wilma never works more than 40 hours in a week. Complete an IPO chart for this problem. Desk-check the algorithm using $10 as the hourly pay and 35 as the number of hours worked. Then desk-check it using $15 as the hourly pay and 25 as the number of hours worked.

INTRODUCTORY

5. Jenna Williams is paid based on an annual salary rather than an hourly wage. She wants a program that calculates the amount of her gross pay for each pay period. Complete an IPO chart for this problem. Desk-check the algorithm using $35,000 as the annual salary and 52 as the number of pay periods. Then desk-check it using $50,000 as the annual salary and 24 as the number of pay periods.

INTRODUCTORY

6. Rent A Van wants a program that calculates the total cost of renting a van. Customers pay a base fee plus a charge per mile driven. Complete an IPO chart for this problem. Desk-check the algorithm using $50, $.20, and 1000 as the base fee, charge per mile, and number of miles driven. Then desk-check it using your own set of data.

INTERMEDIATE

7. The accountant at Typing Haven wants a program that will help her prepare a customer's bill. She will enter the number of typed envelopes and the number of typed pages, as well as the charge per typed envelope and the charge per typed page. The program should calculate and display the amount due for the envelopes, the amount due for the pages, and the total amount due. Complete an IPO chart for this problem. Desk-check the algorithm using 50, 100, $.10, and $.25 as the number of typed envelopes, the number of typed pages, the charge per typed envelope, and the charge per typed page. Then desk-check it using your own set of data.

INTERMEDIATE

8. The Paper Tree store wants a program that calculates and displays the number of single rolls of wallpaper needed to cover a room. The salesclerk will provide the room's length, width, and ceiling height, in feet. He or she also will provide the number of square feet a single roll will cover. Complete an IPO chart for this problem. Desk-check the algorithm using 10, 12, 8, and 30 as the room's length, width, ceiling height, and number of square feet a single roll will cover. Then desk-check it using your own set of data.

ADVANCED

ADVANCED

9. The payroll clerk at Nosaki Company wants a program that calculates and displays an employee's gross pay, federal withholding tax (FWT), Social Security and Medicare (FICA) tax, state tax, and net pay. The clerk will enter the hours worked (which is never over 40), hourly pay rate, FWT rate, FICA tax rate, and state income tax rate. Complete an IPO chart for this problem. Desk-check the algorithm using 30, $10, .2, .08, and .04 as the hours worked, pay rate, FWT rate, FICA rate, and state tax rate. Then desk-check it using your own set of data.

SWAT THE BUGS

10. GeeBees Clothiers is having a sale. The store manager wants a program that allows a salesclerk to enter the original price of an item and the discount rate. The program should calculate and display the amount of the discount and the sale price. The algorithm in Figure 2-43 is supposed to solve this problem, but it is not working correctly. Correct the algorithm, and then desk-check it using an original price of $100 and a discount rate of .25 (the decimal equivalent of 25%).

Input	Processing	Output
original price discount rate	Processing items: none Algorithm: 1. enter the original price and discount rate 2. calculate the sale price by subtracting the discount from the original price 3. display the discount and price	discount sale price

Figure 2-43

SWAT THE BUGS

11. Etola Systems wants a program that displays the ending inventory amount, given the beginning inventory amount, the amount sold, and the amount returned. The algorithm shown in Figure 2-44 is supposed to solve this problem, but it is not working correctly. First, calculate the expected output using a beginning inventory of 50, an amount sold of 10, and an amount returned of 2. Then use these values to desk-check the algorithm. Correct the algorithm, and then desk-check it again.

Input	Processing	Output
beginning inventory amount sold amount returned	Processing items: none Algorithm: 1. enter the beginning inventory, amount sold, and amount returned 2. calculate the ending inventory by adding the amount sold to the beginning inventory and then subtracting the amount returned from the result 3. display the ending inventory	ending inventory

Figure 2-44

Answers to TRY THIS Exercises

1. See Figures 2-45 and 2-46.

Input	Processing	Output
sale1 sale2 commission rate	Processing items: total sales Algorithm:	commission

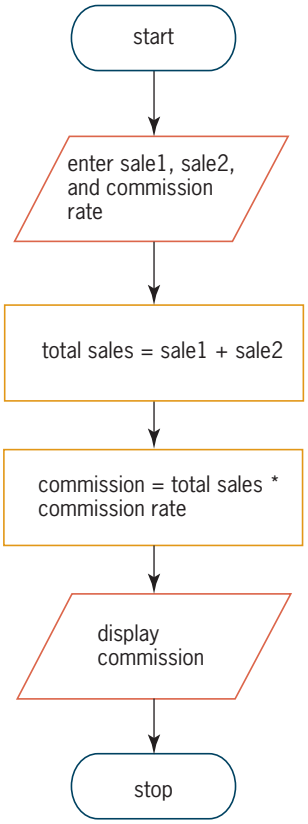


Figure 2-45

sale1	sale2	commission rate	commission
~~1000~~	~~2000~~	~~.05~~	~~150~~
3000	2500	.06	330

Figure 2-46

2. See Figures 2-47 and 2-48.

Input	Processing	Output
cup price plate price cups purchased plates purchased sales tax rate	Processing items: total cup cost total plate cost subtotal Algorithm: 1. enter the cup price, plate price, cups purchased, plates purchased, and sales tax rate 2. calculate the total cup cost by multiplying the cups purchased by the cup price 3. calculate the total plate cost by multiplying the plates purchased by the plate price 4. calculate the subtotal by adding together the total cup cost and total plate cost 5. calculate the total cost by multiplying the subtotal by the sales tax rate and then adding the result to the subtotal 6. display the total cost	total cost

Figure 2-47

cup price	plate price	cups purchased	plates purchased	sales tax rate
~~.50~~	~~1~~	~~35~~	~~35~~	~~.02~~
.25	.75	20	10	.06

total cup cost	total plate cost	subtotal	total cost
~~17.50~~	~~35~~	~~52.50~~	~~53.55~~
5	7.50	12.50	13.25

Figure 2-48

CHAPTER 3

Variables and Constants

After studying Chapter 3, you should be able to:

- Distinguish among a variable, named constant, and literal constant
- Explain how data is stored in memory
- Select an appropriate name, data type, and initial value for a memory location
- Declare a memory location in C++

Beginning Step 4 in the Problem-Solving Process

Chapter 2 covered the first three steps in the problem-solving process, which were to analyze the problem, plan the algorithm, and then desk-check the algorithm. When the programmer is satisfied that the algorithm is correct, he or she moves on to the fourth step, which is to code the algorithm into a program. Coding the algorithm refers to the process of translating the algorithm into a language that the computer can understand; in this book, you will use the C++ programming language. Programmers use the information in the IPO chart as a guide when coding the algorithm. The programmer begins by assigning a descriptive name to each unique input, processing, and output item. The programmer also assigns each item a data type and (optionally) an initial value. The name, data type, and initial value are used to store the input, processing, and output items in the computer's internal memory while the program is running.

Internal Memory

Inside every computer is a component called internal memory. The internal memory of a computer is composed of memory locations, with each memory location having a unique numeric address. It may be helpful to picture memory locations as storage bins, similar to the ones illustrated in Figure 3-1. However, unlike the storage bins shown in the figure, each storage bin (memory location) inside a computer can hold only one item at a time. The item can be a number, such as 5 or 45.89. It also can be **text**, which is a group of characters treated as one unit and not used in a calculation. Examples of text include a name, an address, or a phone number. The item also can be a C++ program instruction. Some of the memory locations inside the computer are automatically filled with data while you use your computer. For example, when you enter the number 5 at your keyboard, the computer saves the number 5 in a memory location for you. Likewise, when you start an application, each program instruction is placed in a memory location, where it awaits processing.

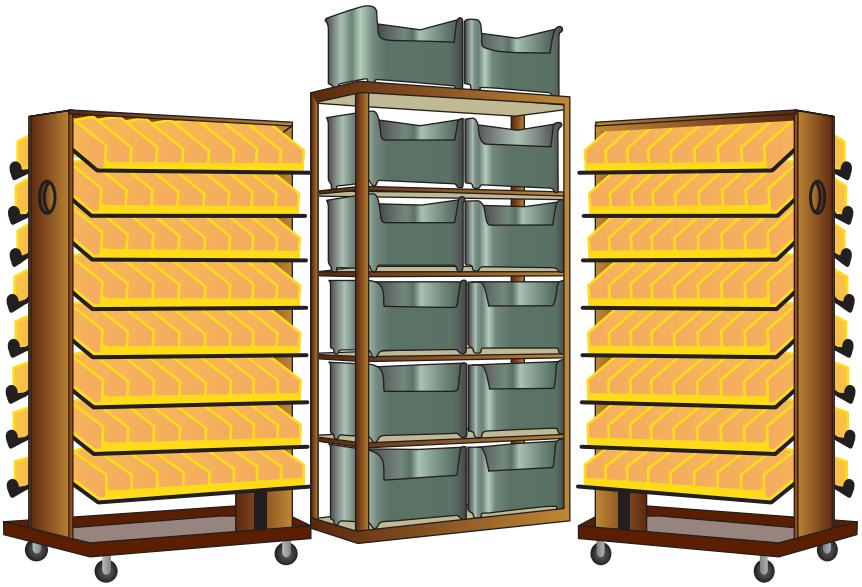


Figure 3-1 Illustration of storage bins

A programmer can reserve memory locations for use in a program. As mentioned earlier, the memory locations will store the values of the input, processing, and output items as the program is running. Reserving a memory location also is referred to as declaring the memory location. You declare a memory location using a C++ instruction that assigns a name, data type, and (optionally) an initial value to the location. The name allows the programmer to refer to the memory location using one or more descriptive words, rather than a cryptic numeric address, in code. The data type indicates the type of data—for example, numeric or textual—the memory location will store. There are two types of memory locations that a programmer can reserve: variables and named constants. A **variable** is a memory location whose value can change (vary) during **runtime**, which is when a program is running. Most of the memory locations declared in a program are variables. A **named constant**, on the other hand, is a memory location whose value cannot be changed during runtime. In a program that inputs the radius of any circle and then calculates and outputs the circle's area, a programmer would declare variables to store the values of the radius and area; doing this allows those values to vary while the program is running. However, he or she would declare a named constant to store the value of pi (π), which is used in the formula for calculating the area of a circle. A named constant is appropriate in this case because the value of pi will always be the same.



The formula for calculating the area of a circle is πr^2. The value of pi (π) rounded to six decimal places is 3.141593.

Selecting a Name for a Memory Location

Every memory location that a programmer declares must be assigned a name. The name, also called the identifier, should be descriptive in that it should help you remember the memory location's purpose. In other words, it should describe the contents of the memory location. A good name for a memory location is one that is meaningful right after you finish a program and also months or years later when you (or perhaps a coworker) need to modify the program. Besides being descriptive, the name must follow several specific rules in C++. The name must begin with a letter and contain only letters, numbers, and the underscore character. No punctuation marks, spaces, or other special characters (such as $ or %) are allowed in the name. In addition, the name cannot be a **keyword**, which is a word that has a special meaning in the programming language you are using. Keywords also are referred to as reserved words. Appendix B in this book contains a listing of the C++ keywords. Finally, memory location names are case sensitive in C++. This means that, in addition to using the exact spelling when referring to a specific memory location in a program, you also must use the exact case. In other words, if you declare a memory location named `discount` at the beginning of a program, you must use the name `discount`, rather than `Discount` or `DISCOUNT`, to refer to the memory location throughout the program.



Refer to the Tip that appears next to Figure 3-2 for an exception to beginning a memory location's name with a letter.



All C++ keywords are entered using lowercase letters.

Many C++ programmers use uppercase letters when naming named constants and use lowercase letters when naming variables. This practice allows them to easily distinguish between the named constants and variables in a program. If a named constant's name contains more than one word, an underscore character can be used to separate the words, like this: TAX_RATE. However, if a variable's name contains two or more words, most C++ programmers enter the name using **camel case**, which means they capitalize the first letter in the second and subsequent words in the name, like this: `grossPay`. Camel case refers to the fact that the uppercase letters appear as "humps" in the name because they are taller than the lowercase letters.

The rules for naming memory locations in C++ are shown in Figure 3-2, along with examples of valid and invalid names.

HOW TO Name a Memory Location in C++

1. The name must begin with a letter.
2. The name can contain only letters, numbers, and the underscore character. No punctuation marks, spaces, or other special characters are allowed in the name.
3. The name cannot be a keyword. Appendix B contains a listing of keywords in C++.
4. Names in C++ are case sensitive.

Valid names

`grossPay`, `interest`, `TAX_RATE`, `PI`

Invalid names	Reason
`2011Sales`	the name must begin with a letter
`end Balance`	the name cannot contain a space
`first.name`	the name cannot contain punctuation
`int`	the name cannot be a keyword
`RATE%`	the name cannot contain a special character



Technically, a memory location's name in C++ can begin with an underscore. However, this usually is done only for the names of memory locations declared within a class.

Figure 3-2 How to name a memory location in C++

Revisiting the Treyson Mobley Problem

In Chapter 2, you analyzed a problem specification involving Treyson Mobley and a waiter's tip. You then planned and desk-checked an appropriate algorithm. The problem specification is shown in Figure 3-3, along with one of the IPO charts and desk-check tables you created.

Problem specification

Treyson Mobley wants a program that calculates and displays the amount he should tip a waiter at a restaurant. The program should subtract any liquor charge from the total bill and then calculate the tip (using a percentage) on the remainder.

Input	Processing	Output
total bill liquor charge tip percentage	Processing items: total bill without liquor charge Algorithm: 1. enter the total bill, liquor charge, and tip percentage 2. calculate the total bill without liquor charge by subtracting the liquor charge from the total bill 3. calculate the tip by multiplying the total bill without liquor charge by the tip percentage 4. display the tip	tip

total bill	liquor charge	tip percentage	total bill without liquor charge	tip
~~45~~	~~10~~	~~.2~~	~~35~~	~~7~~
30	0	.15	30	4.50

Figure 3-3 Problem specification, IPO chart, and desk-check table from Chapter 2

The IPO chart in Figure 3-3 contains a total of five input, processing, and output items. Therefore, five memory locations will be needed to store the values of the items. The memory locations will be variables, because each item's value should be allowed to vary during runtime. Figure 3-4 lists possible names (identifiers) for the variables.

IPO chart item	Variable name
total bill	`totalBill`
liquor charge	`liquor`
tip percentage	`tipPercent`
total bill without liquor charge	`totalNoLiquor`
tip	`tip`

Figure 3-4 Names of the variables for the Treyson Mobley problem

Mini-Quiz 3-1



The answers to Mini-Quiz questions are located in Appendix A.

1. How many items can a memory location store at a time?
2. Which of the following should be used in a C++ program to refer to the `quantity` variable?
   a. `quantity`
   b. `QUANTITY`
   c. `Quantity`
   d. All of the above.
3. Which of the following is a valid name for a memory location?
   a. `jan.Sales`
   b. `2ndQuarterIncome`
   c. `COMMISSION_RATE`
   d. `march$`
4. What are the two types of memory locations that a programmer can reserve?

Selecting a Data Type for a Memory Location

Like storage bins, memory locations come in different types and sizes. The type and size you use depends on the item you want the memory location to store. Some memory locations can store a number, while others can hold text, a date, or a Boolean value (true or false). The item that a memory location will accept for storage is determined by the location's data type, which the programmer assigns to the location when he or she declares it in a program. The most commonly used data types in C++ are listed in Figure 3-5, along with the values each type can store and the amount of memory needed to store a value. Except for the `string` data type, the data types listed in the figure belong to a

The memory requirements and values shown in Figure 3-5 are implementation dependent. The memory requirements listed in the figure are typical for personal computers.

group of data types called fundamental data types. The **fundamental data types** are the basic data types built into the C++ language and often are referred to as primitive data types or built-in data types. The `string` data type, on the other hand, was added to the C++ language through the use of a class and is referred to as a **user-defined data type**. A class is simply a group of instructions that the computer uses to create an object. In this case, the `string` class (user-defined data type) creates a `string` variable, which is considered an object. You will learn more about classes and objects in Chapters 12 through 14.

fundamental data types

user-defined data type

Data type	Stores	Memory required
`short`	an integer Range: –32,768 to 32,767	2 bytes
`int`	an integer Range: –2,147,483,648 to 2,147,483,647	4 bytes
`float`	a real number with 7 digits of precision Range: -3.4×10^{38} to 3.4×10^{38}	4 bytes
`double`	a real number with 15 digits of precision Range: -1.7×10^{308} to 1.7×10^{308}	8 bytes
`bool`	a Boolean value (either `true` or `false`)	1 byte
`char`	one character	1 byte
`string`	zero or more characters	1 byte per character

Figure 3-5 Most commonly used data types in C++

Some programmers pronounce `char` as "care" because it is short for *character*, while others pronounce `char` as in the first syllable of the word *charcoal*.

As Figure 3-5 indicates, `bool` memory locations can store either the Boolean value `true` or the Boolean value `false`. The Boolean values are named in honor of the English mathematician George Boole (1815-1864), who invented Boolean algebra. You could use a `bool` variable in a program to keep track of whether a customer's bill is paid (`true`) or not paid (`false`). Memory locations assigned the `char` data type can store one character only, while `string` memory locations can store zero or more characters. A **character** is a letter, a symbol, or a number that will not be used in a calculation. Memory locations assigned either the `short` or `int` data type can store integers only. An **integer** is a whole number, which is a number that does not contain a decimal place. Examples of integers include the numbers 0, 45, and -678. The differences between the `short` and `int` data types are in the range of numbers each type can store and the amount of memory needed to store the number. Memory locations assigned either the `float` or `double` data type can store **real numbers**, which are numbers that contain a decimal place. Examples of real numbers include the numbers 75.67, -3.45, and 783.5689. The differences between the `float` and `double` data types are in the range of numbers each type can store, the precision with which the number is stored, and the amount of memory needed to store the number. In most of the programs you create in this book, you will use the `int` data type for memory locations that will store integers and use the `double` data type for memory locations that will store numbers with a decimal place. The `double` data type was chosen over the `float` data type because it stores real numbers more precisely, using 15 digits of precision rather than only 7 digits. At this point, however, it is important to

caution you about real numbers. Even with 15 digits of precision, not all real numbers can be represented exactly within the computer's internal memory. As a result, some calculations may not result in accuracy to the penny. You will learn more about using real numbers in calculations in Chapter 4. Figure 3-6 shows the data type selected for each variable in the Treyson Mobley problem. The `double` data type was selected because it allows each variable to store a real number with the greatest precision.

IPO chart item	Variable name	Data type
total bill	`totalBill`	`double`
liquor charge	`liquor`	`double`
tip percentage	`tipPercent`	`double`
total bill without liquor charge	`totalNoLiquor`	`double`
tip	`tip`	`double`

Figure 3-6 Data type assigned to each variable for the Treyson Mobley problem

How Data Is Stored in Internal Memory

Knowing how data is stored in the computer's internal memory will help you understand the importance of a memory location's data type. Numbers are represented in internal memory using the binary (or *base 2*) number system. The **binary number system** uses only the two digits 0 and 1. Although the binary number system may not be as familiar to you as the **decimal number system**, which uses the ten digits 0 through 9, it is just as easy to understand. First, we'll review the decimal (or *base 10*) number system that you learned about in elementary school. Figure 3-7 illustrates how the system works. As the figure indicates, the position of each digit in the decimal number system is associated with the system's base number, 10, raised to a power. Starting with the rightmost position, the positions represent the number 10 raised to a power of 0, 1, 2, 3, and so on. In the decimal number 110, the 0 is in the 10^0 position, the middle 1 is in the 10^1 position, and the leftmost 1 is in the 10^2 position. Keep in mind that, in all numbering systems, the result of raising the base number to the 0th power is 1, and the result of raising it to the 1st power is the base number itself. A base number raised to the 2nd power indicates that the base number should be squared—in other words, multiplied by itself. As a result, the decimal number 110 means zero 1s (10^0), one 10 (10^1), and one 100 (10^2). The decimal number 3475 means five 1s (10^0), seven 10s (10^1), four 100s (10^2), and three 1000s (10^3), and the decimal number 21509 means nine 1s (10^0), zero 10s (10^1), five 100s (10^2), one 1000 (10^3), and two 10000s (10^4).

HOW TO Use the Decimal (*Base 10*) Number System

Decimal number	10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0
110						1	1	0
3475					3	4	7	5
21509				2	1	5	0	9

Figure 3-7 How to use the decimal (*base 10*) number system

Compare the decimal number system illustrated in Figure 3-7 with the binary number system illustrated in Figure 3-8. Like the decimal number system, the position of each digit in the binary number system also is associated with the system's base number raised to a power. However, in the binary number system, the base number is 2 rather than 10. Starting with the rightmost position, the positions represent 2 raised to a power of 0, 1, 2, 3, and so on. In the binary number 110, the 0 is in the 2^0 position, the middle 1 is in the 2^1 position, and the leftmost 1 is in the 2^2 position. Therefore, the binary number 110 means zero 1s (2^0), one 2 (2^1), and one 4 (2^2). The decimal equivalent of the binary number 110 is 6, which is calculated by adding together 0 + 2 + 4 (zero 1s + one 2 + one 4). In other words, the decimal number 6 is stored in a memory location using the binary number 110. The binary number 11010 means zero 1s (2^0), one 2 (2^1), zero 4s (2^2), one 8 (2^3), and one 16 (2^4). The decimal equivalent of the binary number 11010 is 26, which is calculated by adding together 0 + 2 + 0 + 8 + 16. The decimal equivalent of the last binary number shown in Figure 3-8 is 9 (one 1 + zero 2s + zero 4s + one 8).

HOW TO Use the Binary (*Base 2*) Number System

Binary number	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Decimal equivalent
110						1	1	0	6
11010				1	1	0	1	0	26
1001					1	0	0	1	9

Figure 3-8 How to use the binary (*base 2*) number system

Unlike numeric data, character data (which is data assigned to memory locations that can store characters) is represented in internal memory using ASCII codes. **ASCII** (pronounced *ASK-ee*) stands for American Standard Code for Information Interchange. The ASCII coding scheme assigns a specific code to each character on your keyboard. Figure 3-9 shows a partial listing of the ASCII codes along with their binary representations. The full ASCII chart is contained in Appendix C in this book. As Figure 3-9 indicates, the uppercase letter A is assigned the ASCII code 65, which is stored in internal memory using the eight bits ("binary digits") 01000001 (one 64 and one 1). Notice that the lowercase version of each letter is assigned a different ASCII code than the letter's uppercase version. The lowercase letter a, for example, is assigned the ASCII code 97, which is stored in internal memory using the eight bits 01100001. This fact indicates that the computer does not consider both cases of a letter to be equivalent. In other words, the uppercase letter A is not the same as the lowercase letter a. This concept will become important when you compare characters in later chapters.

Character	ASCII	Binary	Character	ASCII	Binary	Character	ASCII	Binary
0	48	00110000	K	75	01001011	g	103	01100111
1	49	00110001	L	76	01001100	h	104	01101000
2	50	00110010	M	77	01001101	i	105	01101001
3	51	00110011	N	78	01001110	j	106	01101010
4	52	00110100	O	79	01001111	k	107	01101011
5	53	00110101	P	80	01010000	l	108	01101100
6	54	00110110	Q	81	01010001	m	109	01101101
7	55	00110111	R	82	01010010	n	110	01101110
8	56	00111000	S	83	01010011	o	111	01101111
9	57	00111001	T	84	01010100	p	112	01110000
:	58	00111010	U	85	01010101	q	113	01110001
;	59	00111011	V	86	01010110	r	114	01110010
A	65	01000001	W	87	01010111	s	115	01110011
B	66	01000010	X	88	01011000	t	116	01110100
C	67	01000011	Y	89	01011001	u	117	01110101
D	68	01000100	Z	90	01011010	v	118	01110110
E	69	01000101	a	97	01100001	w	119	01110111
F	70	01000110	b	98	01100010	x	120	01111000
G	71	01000111	c	99	01100011	y	121	01111001
H	72	01001000	d	100	01100100	z	122	01111010
I	73	01001001	e	101	01100101			
J	74	01001010	f	102	01100110			

Figure 3-9 Partial ASCII chart

At this point, you may be wondering why the numeric characters on your keyboard are assigned ASCII codes. For example, shouldn't a 9 be stored using the binary number system, as you learned earlier? The answer is that the computer uses the binary number system to store the *number* 9, but it uses the ASCII coding scheme to store the *character* 9. But how does the computer know whether the 9 is a number or a character? The answer to this question is simple: by the memory location's data type. Consider a program that displays the message "Enter your pet's age:" on the computer screen. The program stores your response in a variable named `age`. When you press the 9 key on your keyboard in response to the message, the computer uses the data type of the `age` variable to determine whether to store the 9 as a number (using the binary number system) or as a character (using the ASCII coding scheme). If the variable's data type is `int`, the 9 is stored as the binary number 1001 (one 1 + one 8). If the variable's data type is `char`, on the other hand, the 9 is stored as a character using the ASCII code 57, which is represented in internal memory as 00111001 (one 1 + one 8 + one 16 + one 32). The memory location's data type also determines how the computer interprets a memory location's existing data.

If a program instruction needs to access the value stored in a memory location—perhaps to display the value on the screen—the computer uses the memory location's data type to determine the value's data type. To illustrate this point, assume that a memory location named `inputItem` contains the eight bits 01000001. If the memory location's data type is `char`, the computer displays the uppercase letter A on the screen. This is because the computer interprets the 01000001 as the ASCII code 65, which is equivalent to the uppercase letter A. However, if the memory location's data type is `int`, the computer displays the number 65 on the screen, because the 01000001 is interpreted as the binary representation of the decimal number 65. In summary, the data type of a memory location is important because it determines how the data is stored when first entered into the memory location. It also determines how the data is interpreted when the memory location is used in an instruction later in the program.



The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 3-2

1. The `string` data type is one of the fundamental data types in C++.
   a. True
   b. False

2. In the binary number system, the decimal number 32 is represented as _____.
   a. 10000
   b. 10001
   c. 100000
   d. none of the above

3. What is the ASCII code for the lowercase letter b, and how is it represented in the computer's internal memory?

4. Which data type can store a real number?
   a. `double`
   b. `int`
   c. `float`
   d. both a and c

Selecting an Initial Value for a Memory Location

In addition to assigning a name and data type to each variable and named constant used in a program, you also should assign an initial value to each. Assigning an initial (or beginning) value to a memory location is referred to as **initializing**. With the exception of a `bool` memory location, which is initialized using either the C++ keyword `true` or the C++ keyword `false`,

you typically initialize a memory location by assigning a literal constant to it. Unlike variables and named constants, literal constants are not memory locations. Rather, a **literal constant** is an item of data that can appear in a program instruction and be stored in a memory location. The data type of the literal constant should match the data type of the memory location to which it is assigned. Integers should be assigned to memory locations having the `short` or `int` data type. Memory locations having the `float` or `double` data type should be initialized using real numbers. Integers and real numbers are called **numeric literal constants**; examples include the numbers 146, 0.0, and –2.5. Numeric literal constants can contain numbers, the plus sign, the minus sign, and the decimal point. They cannot contain a space, a comma, or a special character, such as the dollar sign ($) or percent sign (%). A numeric literal constant with no decimal place is considered an `int` data type in C++, whereas a numeric literal constant with a decimal place is considered a `double` data type. Programmers use character literal constants to initialize `char` memory locations. A **character literal constant** is one character enclosed in single quotation marks, such as the letter 'X', the dollar sign '$', and a space ' ' (two single quotation marks with a space character between). A `string` memory location is initialized using a **string literal constant**, which is zero or more characters enclosed in double quotation marks. The word "Hello", the message "Enter your pet's age:", and the **empty string** "" (two double quotation marks with no space between) are examples of string literal constants.



A numeric literal constant also can contain either the letter e or the letter E. The letters represent exponential notation, often referred to as e notation. Scientific programs use e notation to represent very small and very large numbers.

When a program instructs the computer to assign a value to a memory location, the computer first compares the value's data type with the memory location's data type. The comparison is made to verify that the value is appropriate for the memory location. If the value's data type does not match the memory location's data type, the computer uses a process called **implicit type conversion** to convert the value to fit the memory location. For example, if a program initializes a `double` variable named `price` to the integer 9, the computer converts the integer to a real number before storing the value in the variable. The computer does this by appending a decimal point and the number 0 to the end of the integer, like this: 9.0. The computer then stores the real number 9.0 in the `price` variable. When a value is converted from one data type to another data type that can store larger numbers, the value is said to be **promoted**. In this case, the `int` value 9 is promoted to the `double` value 9.0. (As shown earlier in Figure 3-5, the `double` data type can store larger numbers than can the `int` data type.) In most cases, the implicit promotion of values does not adversely affect a program's output. However, now consider a program that declares an `int` named constant called `MIN_WAGE`. If you use a real number (such as 7.25) to initialize the named constant, the computer converts the real number to an integer before storing the value in the memory location. The computer does this by truncating (dropping off) the decimal portion of the number. In this case, the computer converts the real number 7.25 to the integer 7. As a result, the number 7 rather than the number 7.25 is assigned to the `MIN_WAGE` named constant. When a value is converted from one data type to another data type that can store only smaller numbers, the value is said to be **demoted**. In this case, the `double` value 7.25 is demoted to the `int` value 7. The implicit demotion of values can adversely affect a program's output. Therefore, it's important to initialize memory locations using values that have the same data type as the memory location.



Although initializing variables is optional in most programming languages, including C++, it is considered a good programming practice to do so and is highly recommended.



Recall that a numeric literal constant with a decimal place is treated as a `double` number.

If a memory location is a named constant, the problem specification and IPO chart will provide the appropriate initial value to use, and that value will remain the same during runtime. (Recall that the contents of a named constant cannot change while the program is running.) The initial value for a variable, on the other hand, is not stated in a problem description or IPO chart, because the user supplies the value while the program is running. As a result, `short` and `int` variables generally are initialized to the integer 0, while `float` and `double` variables are assigned the real number 0.0. Variables declared using the `string` data type usually are initialized to the empty string (""), and `char` variables are initialized to a space (' '). As mentioned earlier, the C++ keywords `true` and `false` are used to initialize `bool` variables. Figure 3-10 shows the initial values for the variables in the Treyson Mobley problem.

IPO chart item	Variable name	Data type	Initial value
total bill	`totalBill`	`double`	`0.0`
liquor charge	`liquor`	`double`	`0.0`
tip percentage	`tipPercent`	`double`	`0.0`
total bill without liquor charge	`totalNoLiquor`	`double`	`0.0`
tip	`tip`	`double`	`0.0`

Figure 3-10 Initial values for the variables in the Treyson Mobley problem

Declaring a Memory Location

Now that you know how to select an appropriate name, data type, and initial value for a memory location, you can learn how to declare variables and named constants in a C++ program. We'll begin with variables. You declare a variable using a **statement**, which is a C++ instruction that causes the computer to perform some action after being executed (processed) by the computer. A statement that declares a variable, for example, causes the computer to set aside a memory location with the name, data type, and initial value you provide. A variable declaration statement is one of many different types of statements in C++. The syntax and examples of a variable declaration statement are shown in Figure 3-11. The term **syntax** refers to the rules of a programming language. One rule in C++ is that all statements must end with a semicolon; the syntax and examples in Figure 3-11 follow this rule. Another rule is that the programmer must provide a data type and name for the variable being declared. He or she also can provide an initial value for the variable. Items that the programmer provides are italicized in a statement's syntax, as shown in Figure 3-11. Items appearing in square brackets in a syntax—in this case, the = symbol and *initialValue*—are optional. In other words, the C++ language does not require variables to be initialized. However, initializing variables is highly recommended. If you do not provide an initial value, the variable may contain a meaningless value. Programmers refer to the meaningless value as *garbage*, because it is the remains of what was last stored in the memory location that the variable now occupies. Items in boldface in a syntax are required. In a variable declaration statement, the semicolon is required; the = symbol is required only when the programmer is providing an initial value for the variable.



Using the storage bin analogy from the beginning of the chapter, initializing a variable is similar to removing any junk (or garbage) from a bin before using it.

HOW TO Declare a Variable in C++

Syntax
dataType variableName [= *initialValue*];

Examples
```
int age = 0;
double price = 0.0;
bool paid = false;
char grade = ' ';
string company = "";
```

Figure 3-11 How to declare a variable in C++

After a variable is declared, you then can use its name to refer to it later in the program, such as in a statement that displays the variable's value or uses the value in a calculation. You will learn how to write such statements in Chapter 4. Figure 3-12 shows the declaration statements you would use to declare the five variables in the Treyson Mobley problem.

IPO chart item	Variable name	Data type	Initial value	C++ statement
total bill	totalBill	double	0.0	double totalBill = 0.0;
liquor charge	liquor	double	0.0	double liquor = 0.0;
tip percentage	tipPercent	double	0.0	double tipPercent = 0.0;
total bill without liquor charge	totalNoLiquor	double	0.0	double totalNoLiquor = 0.0;
tip	tip	double	0.0	double tip = 0.0;

Figure 3-12 C++ declaration statements for the variables in the Treyson Mobley problem

Figure 3-13 shows the syntax and examples of a statement that declares a named constant. Recall that italicized items in a syntax indicate information that the programmer must supply. In a named constant declaration statement, the programmer must supply the constant's data type, name, and initial value. Items in boldface in a syntax—in this case, the keyword `const`, the = symbol, and the semicolon—are required. The `const` keyword indicates that the memory location being declared is a named constant, which means its value cannot be changed during runtime. If a program statement attempts to change the value stored in a named constant, the C++ compiler will display an error message. As you learned in Chapter 1, a compiler converts the instructions written in a high-level language (such as C++) into the 0s and 1s the computer can understand.

HOW TO Declare a Named Constant in C++

Syntax

const *dataType constantName* **=** *value*;

Examples

```
const double PI = 3.141593;
const int MIN_AGE = 65;
const bool INSURED = true;
const char YES = 'Y';
const string BANK = "Harrison Trust and Savings";
```

Figure 3-13 How to declare a named constant in C++

As you can with variables, you can use a named constant in another statement that appears after its declaration statement. For example, after entering the `const double PI = 3.141593;` statement in a program, you can use `PI` in a statement that calculates the area of a circle; the computer will use the value stored in the named constant (3.141593) to calculate the area. Using named constants in a program has several advantages. First, named constants make a program more self-documenting and easier to modify, because they allow the use of meaningful words in place of values that are less clear. The named constant `PI`, for example, is much more meaningful than the number 3.141593, which is the value of pi rounded to six decimal places. Second, unlike the value stored in a variable, the value stored in a named constant cannot be inadvertently changed during runtime. Third, typing `PI` rather than 3.141593 in a statement is easier and less prone to typing errors. If you do mistype `PI` in the area calculation statement—for example, if you type `Pi` rather than `PI`—the C++ compiler will display an error message. Mistyping 3.141593 in the area calculation statement, however, will not trigger an error message and will result in an incorrect answer. Finally, if a named constant's value needs to be changed in the future, you will need to modify only the declaration statement, rather than all of the statements that use the value.



The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 3-3

1. Which of the following is a character literal constant?
   a. '56'
   b. '%'
   c. "a"
   d. both a and b

2. Which of the following is a string literal constant?
   a. "$"
   b. "Good Morning!"
   c. ""
   d. all of the above

3. If you assign the number 10.25 to an `int` variable, what will the computer store in the variable?

4. Write a C++ statement that declares and initializes an `int` variable named `population`.

5. Write a C++ statement that declares the `MAX_PAY` named constant. The constant should have the `double` data type and contain the number 25.55.



LAB 3-1 Stop and Analyze

Study the problem specification, IPO chart, and desk-check table shown in Figure 3-14 and then answer the questions.



The answers to the labs are located in Appendix A.

Problem specification
Aiden Nelinski is paid every Friday. He is scheduled to receive anywhere from a 2% to 4.5% raise next week. He wants a program that calculates and displays the amount of his new weekly pay.

Input	Processing	Output
current weekly pay raise percentage	Processing items: none Algorithm: 1. enter the current weekly pay and raise percentage 2. calculate the new weekly pay by multiplying the current weekly pay by the raise percentage and then adding the result to the current weekly pay 3. display the new weekly pay	new weekly pay

current weekly pay	raise percentage	new weekly pay
~~300~~	~~.02~~	~~306~~
500	.025	512.50

Figure 3-14 Problem specification, IPO chart, and desk-check table for Lab 3-1

QUESTIONS

1. How many memory locations will the Aiden Nelinski problem require?
2. How many of the memory locations will be variables, and how many will be named constants? Why did you choose one type over the other?
3. If the input and output items are real numbers, which data types could be assigned to the memory locations that store these items?

4. How would you write the appropriate declaration statements? Use the `double` data type and the names `currentPay`, `raiseRate`, and `newPay`.
5. If the raise percentage was always 2%, how would you declare the memory location that will store it? Use the `double` data type and the name `RAISE_RATE`.



LAB 3-2 Plan and Create

In this lab, you will plan and create an algorithm that displays the area of a circle. The problem specification is shown in Figure 3-15.

> Professor Chang wants a program that calculates and displays the area of a circle, given the circle's radius. The formula for calculating the area of a circle is πr^2, where π and r represent pi and the radius, respectively. The professor wants to use the value of pi rounded to two decimal places, which is 3.14.

Figure 3-15 Problem specification for Lab 3-2

First, analyze the problem, looking for the output first and then for the input. Recall that the output answers the question *What does the user want to see displayed on the screen, printed on paper, or stored in a file?*, and the input answers the question *What information will the computer need to know to display, print, or store the output items?* In this case, the user wants to see the circle's area displayed on the screen. To do this, the computer will need to know the circle's radius and the value of pi. The radius will be entered by the user, whereas the problem specification indicates that the value to use for pi is 3.14. Figure 3-16 shows the input and output items entered in an IPO chart.

Input	Processing	Output
radius pi (3.14)	Processing items: Algorithm:	area

Figure 3-16 Partially completed IPO chart showing the input and output items

After determining a problem's output and input, you then plan its algorithm. Recall that most algorithms begin with an instruction to enter the input items into the computer, followed by instructions that process the input items, typically including the items in one or more calculations. Most algorithms end with one or more instructions that display, print, or store the output items. Figure 3-17 shows the completed IPO chart for the circle area problem.

Input	Processing	Output
radius pi (3.14)	Processing items: none Algorithm: 1. enter the radius 2. calculate the area by multiplying the radius by itself and then multiplying the result by pi 3. display the area	area

Figure 3-17 Completed IPO chart for Lab 3-2

After completing the IPO chart, you then move on to the third step in the problem-solving process, which is to desk-check the algorithm. You begin by choosing a set of sample data for the input values. You then use the values to manually compute the expected output. You will desk-check the current algorithm twice: first using 4 as the radius and then using 5.5. For the first desk-check, the area should be 50.24. For the second desk-check, the area should be 94.985. The manual calculations for both desk-checks are shown in Figure 3-18.

First desk-check		Second desk-check	
4	(radius)	5.5	(radius)
* 4	(radius)	* 5.5	(radius)
* 3.14	(pi)	* 3.14	(pi)
50.24	(area)	94.985	(area)

Figure 3-18 Manual area calculation for the two desk-checks

Next, you create a desk-check table that contains one column for each input, processing, and output item. You then begin desk-checking the algorithm. Figure 3-19 shows the completed desk-check table. Notice that the amounts in the area column agree with the results of the manual calculations shown in Figure 3-18.

radius	pi	area
~~4~~	~~3.14~~	~~50.24~~
5.5	3.14	94.985

Figure 3-19 Completed desk-check table for Lab 3-2

After desk-checking an algorithm to ensure that it works correctly, you can begin coding it. You begin by declaring memory locations that will store the values of the input, processing (if any), and output items. The circle area problem will need three memory locations to store the values of the radius, pi, and area. The radius and area values should be stored in variables, because the user should be allowed to change the radius value, which then will change the area value, while the program is running. The value of pi, however, will be stored in a named constant, because its value should not change during runtime. The variables and named constant will store real numbers, so you will use the `double` data type for each one. Figure 3-20 shows the input, processing, and output items from the IPO chart, along with the corresponding C++ statements.

IPO chart information	C++ instructions
Input	
radius	`double radius = 0.0;`
pi (3.14)	`const double PI = 3.14;`
Processing	
none	
Output	
area	`double area = 0.0;`

Figure 3-20 IPO chart information and C++ instructions for Lab 3-2



LAB 3-3 Modify

Modify the IPO chart shown earlier in Figure 3-17 so that it includes the radius squared as a processing item. Then make the appropriate modifications to Figure 3-20.



LAB 3-4 Desk-Check

Using the IPO chart modified in Lab 3-3, make the appropriate modifications to the manual calculations and desk-check table shown earlier in Figures 3-18 and 3-19.



LAB 3-5 Debug

Correct the C++ instructions shown in Figure 3-21. The memory locations will store real numbers.

IPO chart information	C++ instructions
Input	
first number	`first = 0.0;`
second number	`second = 0.0;`
third number	`third = 0.0;`
Processing	
sum	`sum = 0.0`
Output	
average	`average = 0.0;`

Figure 3-21 IPO chart information and C++ instructions for Lab 3-5

Summary

- The fourth step in the problem-solving process is to code the algorithm, which means to translate it into a language that the computer can understand. You begin by declaring a memory location for each unique input, processing, and output item listed in the IPO chart. The memory locations will store the values of those items while the program is running.
- Numeric data is stored in the computer's internal memory using the binary number system. Character data is stored using the ASCII coding scheme.
- A memory location can store only one value at a time.
- A memory location's data type determines how a value is stored in the location, as well as how the value is interpreted when retrieved.
- Programmers can declare two types of memory locations: variables and named constants. You declare a memory location using a statement that assigns a name, data type, and initial value to the memory location. The initial value is required when declaring named constants but is optional when declaring variables. However, it is highly recommended that variables be initialized to ensure they don't contain garbage.
- In most cases, memory locations are initialized using a literal constant. The exception to this is a `bool` memory location, which is initialized using a keyword (`true` or `false`).
- The data type of a literal constant assigned to a memory location should be the same as the memory location's data type. If the data types do not match, the computer uses implicit type conversion to either promote or demote the value to fit the memory location. Promoting a value does not usually affect a program's output; however, demoting a value may cause a program's output to be incorrect.
- The C++ programming language has a set of rules, called syntax, which you must follow when using the language. One rule is that all statements in C++ must end with a semicolon.

Key Terms

ASCII—a coding scheme used to represent character data; stands for American Standard Code for Information Interchange

Binary number system—a system that uses only the two digits 0 and 1; used to represent numeric data in the computer's internal memory

Camel case—a naming convention that capitalizes only the first letter in the second and subsequent words in a memory location's name

Character—a letter, a symbol, or a number that will not be used in a calculation

Character literal constant—one character enclosed in single quotation marks

Decimal number system—a system that represents numbers using the digits 0 through 9

Demoted—refers to the conversion of a number from one data type to another data type that can store only smaller numbers

Empty string—two quotation marks with no space between, like this “”

Fundamental data types—the basic data types built into the C++ language; also called primitive data types or built-in data types

Implicit type conversion—the process the computer follows when converting a numeric value to fit a memory location that has a different data type

Initializing—assigning a beginning value to a memory location

Integer—a whole number, which is a number without a decimal place

Keyword—a word that has a special meaning in the programming language you are using

Literal constant—an item of data that can appear in a program instruction and be stored in a memory location

Named constant—a memory location whose value cannot be changed while a program is running

Numeric literal constants—numbers

Promoted—refers to the conversion of a number from one data type to another data type that can store larger numbers

Real numbers—numbers that contain a decimal place

Runtime—occurs while a program is running

Statement—a C++ instruction that causes the computer to perform some action after being executed (processed) by the computer; all statements in C++ must end with a semicolon

String literal constant—zero or more characters enclosed in double quotation marks

Syntax—the rules you need to follow when using a programming language; every programming language has its own syntax

Text—a group of characters treated as one unit and not used in a calculation

User-defined data type—a data type added to the C++ language through the use of a class; an example is the `string` data type

Variable—a memory location whose value can change (vary) while a program is running

Review Questions

1. The rules you must follow when using a programming language are called its _____.

   a. guidelines

   b. procedures

   c. regulations

   d. syntax

2. Which of the following declares a variable that can store a real number?

   a. `commission double = 0.0;`

   b. `double commission = 0.0`

   c. `double commission = 0.0;`

   d. `commission = 0.0;`

3. A C++ statement must end with a _____.

   a. colon

   b. comma

   c. period

   d. semicolon

4. The declaration statement for a named constant requires _____.

   a. a data type

   b. a name

   c. a value

   d. all of the above

5. Which of the following creates a variable that can store whole numbers only?

   a. `int numItems = 0;`

   b. `int numItems = '0';`

   c. `int numItems = "0";`

   d. `integer numItems = 0;`

6. Which of the following is a valid name for a variable?

   a. `amount-sold`

   b. `amountSold`

   c. `1stQtrAmountSold`

   d. both b and c

7. Which of the following declares a `char` named constant called `TOP_GRADE`?

   a. `const char TOP_GRADE = 'A';`

   b. `const char TOP_GRADE = "A";`

   c. `const char TOP_GRADE;`

   d. both a and c

8. A memory location contains the following eight bits: 00110111. If the memory location's data type is `int`, the computer will interpret the eight bits as the number _____.

   a. 7

   b. 55

   c. 56

   d. 110, 111

9. A memory location contains the following eight bits: 00110111. If the memory location's data type is `char`, the computer will interpret the eight bits as the character _____.

   a. 7

   b. 55

   c. 56

   d. none of the above

10. If you use a real number to initialize an `int` variable, the real number will be _____ before it is stored in the variable.

    a. demoted

    b. promoted

    c. reduced

    d. upgraded

Exercises



Pencil and Paper

TRY THIS

1. Complete the C++ instructions column in Figure 3-22. Use the `double` data type for the memory locations. (The answers to TRY THIS Exercises are located at the end of the chapter.)

IPO chart information	C++ instructions
Input	
sale1	
sale2	
commission rate	
Processing	
total sales	
Output	
commission	

Figure 3-22

TRY THIS

2. Complete the C++ instructions column in Figure 3-23. The numbers of cups and plates purchased will be integers. All of the remaining items will be real numbers. Use the `int` and `double` data types. (The answers to TRY THIS Exercises are located at the end of the chapter.)

IPO chart information	C++ instructions
Input	
cup price	
plate price	
cups purchased	
plates purchased	
sales tax rate (5.5%)	
Processing	
total cup cost	
total plate cost	
subtotal	
Output	
total cost	

Figure 3-23

MODIFY THIS

3. Complete TRY THIS Exercise 1, and then modify the IPO chart information and C++ instructions to indicate that the commission rate will always be 10%.

INTRODUCTORY

4. Chris Johanson wants a program that calculates and displays a 10%, 15%, and 20% tip on his total restaurant bill. First, create an IPO chart for this problem, and then desk-check the algorithm twice, using $35.80 and $56.78 as the total bill. After desk-checking the algorithm, list the input, processing, and output items in a chart similar to the one shown in Figure 3-23, and then enter the appropriate C++ declaration statements.

INTRODUCTORY

5. Tile Limited wants a program that allows its salesclerks to enter the length and width (both in feet) of a rectangle and the price of a square foot of tile. The program should calculate and display the area of the rectangle and the total price of the tile. First, create an IPO chart for this problem, and then desk-check the algorithm twice. For the first desk-check, use 10 feet, 12 feet, and $2.39 as the length, width, and tile price, respectively. For the second desk-check, use 5.5 feet, 10.5 feet, and $3.50. After desk-checking the algorithm, list the input, processing, and output items in a chart similar to the one shown in Figure 3-23, and then enter the appropriate C++ declaration statements.

INTERMEDIATE

6. Builders Inc. wants a program that allows its salesclerks to enter the diameter of a circle and the price of railing material per foot. The program should calculate and display the total price of the railing material. Use 3.1416 as the value of pi. First, create an IPO chart for this problem, and then desk-check the algorithm twice. For the first desk-check, use 35 feet as the diameter and $2 as the price per foot. For the second desk-check, use 15.5 and $3.50. After desk-checking the algorithm, list the input, processing, and output items in a chart similar to the one shown in Figure 3-23, and then enter the appropriate C++ declaration statements.

INTERMEDIATE

7. Currency Traders wants a program that converts American dollars to British pounds, Mexican pesos, and Japanese yen and then displays the results. Use the following conversion rates for one American dollar: 0.571505 British pounds, 10.7956 Mexican pesos, and 112.212 Japanese yen. First, create an IPO chart for this problem, and then desk-check the algorithm twice, using 1000 and 50 as the number of American dollars. After desk-checking the algorithm, list the input, processing, and output items in a chart similar to the one shown in Figure 3-23, and then enter the appropriate C++ declaration statements.

ADVANCED

8. Crispies Bagels and Bites wants a program that allows its salesclerks to enter the number of bagels, donuts, and cups of coffee a customer orders. Bagels are 99¢, donuts are 75¢, and coffee is $1.20 per cup. The program should calculate and display the total price of the customer's order. First, create an IPO chart for this problem, and then desk-check the algorithm twice. For the first desk-check, use 0, 12, and 2 as the number of bagels, donuts, and cups of coffee, respectively. For the second desk-check, use 2, 6, and 1. After desk-checking the algorithm, list the input, processing, and output items in a chart similar to the one shown in Figure 3-23, and then enter the appropriate C++ declaration statements.

9. The payroll clerk at Cartwright Industries wants a program that displays an employee's weekly net pay. The clerk will enter the employee's weekly gross pay. From the gross pay, the program will need to subtract the appropriate federal and state taxes. Use 20% as the FWT (Federal Withholding Tax) rate, 8% as the FICA (Social Security and Medicare) tax rate, and 4% as the state income tax rate. First, create an IPO chart for this problem, and then desk-check the algorithm twice, using $500 and $235.50 as the weekly gross pay. After desk-checking the algorithm, list the input, processing, and output items in a chart similar to the one shown in Figure 3-23, and then enter the appropriate C++ declaration statements.

ADVANCED

10. Correct the C++ instructions shown in Figure 3-24.

SWAT THE BUGS

IPO chart information	C++ instructions
Input	
original price	`double original = 0.0;`
discount rate (10%)	`double DISC_RATE = 10%;`
Processing	
none	
Output	
discount	`int discount = 0;`
new price	`double new price = 0.0;`

Figure 3-24

Answers to TRY THIS Exercises

1. See Figure 3-25.

IPO chart information	C++ instructions
Input	
sale1	`double sale1 = 0.0;`
sale2	`double sale2 = 0.0;`
commission rate	`double commissionRate = 0.0;`
Processing	
total sales	`double totalSales = 0.0;`
Output	
commission	`double commission = 0.0;`

Figure 3-25

2. See Figure 3-26.

IPO chart information	C++ instructions
Input	
cup price	`double cupPrice = 0.0;`
plate price	`double platePrice = 0.0;`
cups purchased	`int cupsPurchased = 0;`
plates purchased	`int platesPurchased = 0;`
sales tax rate (5.5%)	`const double TAX_RATE = .055;`
Processing	
total cup cost	`double totalCupCost = 0.0;`
total plate cost	`double totalPlateCost = 0.0;`
subtotal	`double subtotal = 0.0;`
Output	
total cost	`double totalCost = 0.0;`

Figure 3-26

CHAPTER 4

Completing the Problem-Solving Process

After studying Chapter 4, you should be able to:

- Get numeric and character data from the keyboard
- Display information on the computer screen
- Write arithmetic expressions
- Type cast a value
- Write an assignment statement
- Code the algorithm into a program
- Desk-check a program
- Evaluate and modify a program

Finishing Step 4 in the Problem-Solving Process

The first three steps in the problem-solving process are to analyze the problem, plan the algorithm, and then desk-check the algorithm; the fourth step is to code the algorithm into a program. As you learned in Chapter 3, the programmer begins the fourth step by declaring a memory location for each unique input, processing, and output item listed in the problem's IPO chart. The memory locations will store the values of those items while the program is running. Recall that each memory location must be assigned a name and data type. If the memory location is a named constant, it also must be assigned a value. Assigning an initial value to a variable is optional but highly recommended to ensure that the variable does not contain garbage. Figure 4-1 shows the problem specification, IPO chart information, and variable declaration statements for the Treyson Mobley problem from Chapter 3. Recall that the `double` data type was selected for each variable because it allows each to store a real number with the greatest precision.

Problem specification
Treyson Mobley wants a program that calculates and displays the amount he should tip a waiter at a restaurant. The program should subtract any liquor charge from the total bill and then calculate the tip (using a percentage) on the remainder.

IPO chart information	**C++ instructions**
Input	
total bill	`double totalBill = 0.0;`
liquor charge	`double liquor = 0.0;`
tip percentage	`double tipPercent = 0.0;`
Processing	
total bill without liquor charge	`double totalNoLiquor = 0.0;`
Output	
tip	`double tip = 0.0;`

Algorithm
1. enter the total bill, liquor charge, and tip percentage
2. calculate the total bill without liquor charge by subtracting the liquor charge from the total bill
3. calculate the tip by multiplying the total bill without liquor charge by the tip percentage
4. display the tip

Figure 4-1 Problem specification, IPO chart information, and variable declaration statements

After declaring the necessary memory locations, the programmer begins coding the algorithm. The first instruction in the algorithm shown in Figure 4-1 is to enter the input items. You will have the user enter the items at the keyboard.

Getting Data from the Keyboard

In C++, you use objects to perform standard input and output operations, such as getting a program's input items and displaying its output items. The objects are called **stream objects** because they handle streams. A **stream** is defined in C++ as a sequence of characters. In this section, you will learn about the standard input stream object, `cin` (pronounced *see in*). The **cin** object tells the computer to pause program execution while the user enters one or more characters at the keyboard; the object temporarily stores the characters as they are typed. The `cin` object typically is used with the **extraction operator** (>>), which extracts (removes) characters from the object and sends them "in" to the computer's internal memory. Figure 4-2 illustrates the relationship among the keyboard, `cin` object, extraction operator, and internal memory. As the figure indicates, the characters you type at the keyboard are sent first to the `cin` object, where they remain until the extraction operator removes them, sending them to the computer's internal memory.

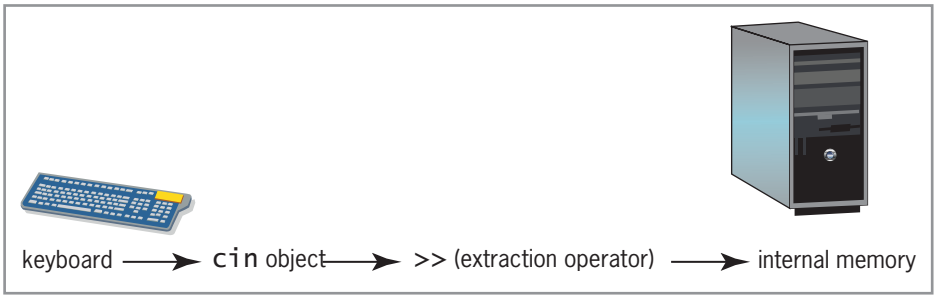


Figure 4-2 Relationship among the keyboard, `cin` object, extraction operator, and internal memory

The extraction operator stops removing characters from the `cin` object when it encounters a **white-space character**, which can be a newline, tab, or blank. You enter a newline character when you press the Enter key on your keyboard. You enter a tab character when you press the Tab key, and you enter a blank character when you press the Spacebar key. Because many strings entered at the keyboard contain one or more blank characters (for example, the string *Raleigh, North Carolina*), the extraction operator is used mainly to get numeric and character data, but not string data. Recall from Chapter 3 that numeric data is a number that will be used in a calculation, while character data is a letter, a symbol, or a number that will not be used in a calculation. String data is zero or more characters treated as one unit. Figure 4-3 shows the syntax and examples of statements that use `cin` and the extraction operator to get numeric and character input. (You will learn how to get string input from the keyboard later in this book.) You can tell that the syntax and examples are statements because a semicolon appears at the end of each. As you learned in Chapter 3, a statement is a C++ instruction that causes the computer to perform some action after being executed (processed) by the computer. The `cin` portion of the `cin >> price;` statement, for example, tells the computer to pause program execution to allow the user to enter the price at the keyboard. The `cin` object temporarily stores the price as the user types it. When the user presses the Enter key, the extraction operator in the statement removes the price from the `cin` object and



The `cin` object and extraction operator allow the user to communicate with the computer while a program is running.

sends it to the computer's internal memory, where it is stored in the `price` variable. Similarly, the `cin >> middleInitial;` statement waits for the user to enter a character and ultimately stores the user's response in the `middleInitial` variable.

HOW TO Use `cin` and >> to Get Numeric or Character Data

Syntax

cin >> *variableName*; — semicolon

Example 1

```
double price = 0.0;
cin >> price;
```

Example 2

```
char middleInitial = ' ';
cin >> middleInitial;
```

Figure 4-3 How to use `cin` and >> to get numeric or character data

The input items in the Treyson Mobley problem are numeric, so you can use the `cin` object and extraction operator to get the items from the user. The appropriate statements are shaded in Figure 4-4.

IPO chart information	**C++ instructions**
Input	
total bill	`double totalBill = 0.0;`
liquor charge	`double liquor = 0.0;`
tip percentage	`double tipPercent = 0.0;`
Processing	
total bill without liquor charge	`double totalNoLiquor = 0.0;`
Output	
tip	`double tip = 0.0;`
Algorithm	

1. enter the total bill, liquor charge, and tip percentage	`cin >> totalBill;`
	`cin >> liquor;`
	`cin >> tipPercent;`

2. calculate the total bill without liquor charge by subtracting the liquor charge from the total bill	
3. calculate the tip by multiplying the total bill without liquor charge by the tip percentage	
4. display the tip	

Figure 4-4 Input statements for the Treyson Mobley problem

When the program is run and the `cin >> totalBill;` statement is processed by the computer, a blank window containing a blinking cursor will appear on the computer screen. The blinking cursor indicates that the computer is waiting for the user to enter *something*, but it does not indicate what that *something* is. Should the user enter an age, price, or middle initial? You indicate the type of information to enter by displaying a message, called a **prompt**, on the computer screen.

Displaying Messages on the Computer Screen

As you learned earlier, you use objects to perform standard input and output operations in C++. In this section, you will learn about the standard output stream object, `cout` (pronounced *see out*). The **cout** object is used with the **insertion operator** (<<) to display information on the computer screen—in other words, to send information "out" to the user. The information can be any combination of literal constants, named constants, or variables. Figure 4-5 shows the syntax and examples of statements that use `cout` and the insertion operator. Notice the required semicolon that appears at the end of the syntax and examples. Also notice that you can include more than one insertion operator in a statement. The `cout << "Enter the price: ";` statement in the figure tells the computer to display a message that prompts the user to enter an item's price. Similarly, the message displayed by the `cout << "What is your middle initial? ";` statement prompts the user to enter his or her middle initial. The `cout` object and insertion operator are not limited to displaying only messages that prompt the user to enter data; you also can use them to display informational messages. The `cout << "End of program";` statement, for instance, alerts the user that the program has ended. The `cout << "Bonus: $" << bonusAmt << endl;` statement, on the other hand, displays a message along with the contents of the `bonusAmt` variable. The `endl` in the statement is one of the stream manipulators in C++. A **stream manipulator** allows you to manipulate (or manage) the characters in either the input or output stream. When used with the `cout` object, the **endl** stream manipulator advances the cursor to the next line on the computer screen, which is equivalent to pressing the Enter key. When typing `endl` (which stands for end of line) in a statement, be sure to type the lowercase letter l rather than the number 1.



The `cout` object and insertion operator allow the computer to communicate with the user while a program is running.



The last example is equivalent to the following three lines of code:

```
cout << "Bonus: $";
cout << bonusAmt;
cout << endl;
```

HOW TO Use the `cout` Object

Syntax

cout << *item1* [**<<** *item2* **<<** *itemN*]**;**

you can use more than one insertion operator

semicolon

Examples

```
cout << "Enter the price: ";
cout << "What is your middle initial? ";
cout << "End of program";
cout << "Bonus: $" << bonusAmt << endl;
```

Figure 4-5 How to use the `cout` object

In order for the user to know what to enter when the Treyson Mobley program is run on the computer, you will display a prompt—one that clearly indicates the information to be entered—for each input item. In this case, you will need three prompts. Each prompt should be entered above its corresponding `cin` statement, because you want the prompt to appear before the user is expected to enter the information. The three prompts are shaded in Figure 4-6. Also shaded in the figure is the statement that displays the tip amount on the computer screen; the statement corresponds to the last instruction in the algorithm. Although it's customary to code an algorithm's instructions in the order they appear in the algorithm, the statement to display the tip amount is included now simply because this section covers displaying messages on the computer screen.



You also can display the tip amount using the following three lines of code:

```
cout << "Tip: $";
cout << tip;
cout << endl;
```

IPO chart information	C++ instructions
Input	
total bill	`double totalBill = 0.0;`
liquor charge	`double liquor = 0.0;`
tip percentage	`double tipPercent = 0.0;`
Processing	
total bill without liquor charge	`double totalNoLiquor = 0.0;`
Output	
tip	`double tip = 0.0;`
Algorithm	

1. enter the total bill, liquor charge, and tip percentage	`cout << "Enter the total bill: ";`
	`cin >> totalBill;`
	`cout << "Enter the liquor charge: ";`
	`cin >> liquor;`
	`cout << "Enter the tip percentage in decimal format: ";`
	`cin >> tipPercent;`

2. calculate the total bill without liquor charge by subtracting the liquor charge from the total bill	
3. calculate the tip by multiplying the total bill without liquor charge by the tip percentage	
4. display the tip	`cout << "Tip: $" << tip << endl;`



The prompts in Figure 4-6 merely display a message on the computer screen. They don't allow the user to actually enter the data being requested; for that, you need the `cin` object and extraction operator.

Figure 4-6 Prompts and output statement for the Treyson Mobley problem



The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 4-1

1. Which of the following stores the value entered at the keyboard in a variable named `grossPay`?

   a. `cin << grossPay;`

   b. `cin >> grossPay;`

c. `cout << grossPay;`

d. `cout >> grossPay;`

2. Which of the following displays the contents of the `grossPay` variable on the computer screen?

   a. `cin << grossPay;`

   b. `cin >> grossPay;`

   c. `cout << grossPay;`

   d. `cout >> grossPay;`

3. Which of the following is considered a white-space character in C++?

   a. a blank

   b. a tab

   c. a newline

   d. all of the above

4. The insertion operator looks like this: _____.

Arithmetic Operators in C++

Instructions 2 and 3 in the algorithm shown in Figure 4-6 involve arithmetic calculations. You direct the computer to perform a calculation by writing an arithmetic expression that contains one or more arithmetic operators. Figure 4-7 lists the standard arithmetic operators available in C++, along with their precedence numbers. The precedence numbers indicate the order in which the computer performs the operation in an expression. Operations with a precedence number of 1 are performed before operations with a precedence number of 2, which are performed before operations with a precedence number of 3, and so on. However, you can use parentheses to override the order of precedence, because operations within parentheses always are performed before operations outside of parentheses.



C++ also provides arithmetic assignment operators, which you will learn about later in this chapter.

Operator	Operation	Precedence number
( )	override normal precedence rules	1
–	negation (reverses the sign of a number)	2
*, /, %	multiplication, division, and modulus arithmetic	3
+, –	addition and subtraction	4

Figure 4-7 Standard arithmetic operators and their order of precedence

Although the negation and subtraction operators listed in Figure 4-7 use the same symbol (a hyphen), there is a difference between both operators: The negation operator is unary, whereas the subtraction operator is binary.

Unary and binary refer to the number of operands required by the operator. Unary operators require one operand, whereas binary operators require two operands. For example, the expression `–7` uses the negation operator to turn the positive number 7 into a negative number. The expression `9 - 4`, on the other hand, uses the subtraction operator to subtract the number 4 from the number 9.

One of the arithmetic operators listed in Figure 4-7, the modulus operator (**%**), might be less familiar to you. The **modulus operator** is used to divide two integers and results in the remainder of the division. For example, the expression `211 % 4` (read 211 mod 4) equals 3, which is the remainder of 211 divided by 4. A common use for the modulus operator is to determine whether a number is even or odd. If you divide a number by 2 and the remainder is 0, the number is even; if the remainder is 1, however, the number is odd.



The modulus operator is used to divide integers only, and the result is always an integer.

Some of the operators listed in Figure 4-7 have the same precedence number. For example, both the addition and subtraction operators have a precedence number of 4. When an expression contains more than one operator having the same priority, those operators are evaluated from left to right. In the expression `5 + 12 / 3 - 1`, for instance, the division (/) is performed first, then the addition (+), and then the subtraction (–). The result of the expression is the number 8, as shown in Figure 4-8. You can use parentheses to change the order in which the operators in an expression are evaluated. For example, as Figure 4-8 shows, the expression `5 + 12 / (3 - 1)` evaluates to 11 rather than to 8. This is because the parentheses tell the computer to perform the subtraction operation first.

Original expression	5 + 12 / 3 – 1
The division is performed first	5 + 4 – 1
The addition is performed next	9 – 1
The subtraction is performed last	8
Original expression	5 + 12 / (3 – 1)
The subtraction is performed first	5 + 12 / 2
The division is performed next	5 + 6
The addition is performed last	11

Figure 4-8 Expressions containing more than one operator having the same precedence

Type Conversions in Arithmetic Expressions

In Chapter 3, you learned about implicit type conversions in statements that declare memory locations. Recall that, if necessary, the computer will either promote or demote the value in a declaration statement to match the memory location's data type. The computer also makes implicit type conversions when processing some arithmetic expressions. More specifically, when performing an arithmetic operation with two values having different data types, the value with the lower-ranking data type is always promoted, temporarily, to the higher-ranking data type. A data type ranks higher than another data type if it can store larger numbers. The value returns to its original data type after the operation is performed. Figure 4-9 shows examples of expressions that require implicit type conversions. The figure also explains how each

expression is evaluated by the computer. In Examples 2 and 4, `firstNum` is an `int` variable that contains the number 5. When a variable name appears in an expression, the computer uses the value stored in the variable when evaluating the expression.



As you learned in Chapter 3, a number with a decimal place is considered a `double` number in C++.



When both operands in an expression are integers, the result is an integer. When both are `double` numbers, the result is a `double` number. When one operand is an integer and the other is a `double` number, the result is a `double` number.

Example 1 `3 * 1.15`
The integer 3 is implicitly promoted to the `double` number 3.0 before being multiplied by the `double` number 1.15. The result is the `double` number 3.45.

Example 2 `9 * (2.5 + firstNum)`
1. The value stored in the `firstNum` variable (the integer 5) is implicitly promoted to the `double` number 5.0 before it is added to the `double` number 2.5. The result is the `double` number 7.5.

2. The integer 9 is implicitly promoted to the `double` number 9.0 before being multiplied by the `double` number 7.5 (the result of Step 1). The result is the `double` number 67.5.

Example 3 `9.8 / 2`
The integer 2 is implicitly promoted to the `double` number 2.0 before it is divided into the `double` number 9.8. The result is the `double` number 4.9.

Example 4 `firstNum / 2.0`
The value stored in the `firstNum` variable (the integer 5) is implicitly promoted to the `double` number 5.0 before being divided by the `double` number 2.0. The result is the `double` number 2.5.

Figure 4-9 Examples of expressions that require implicit type conversions

At this point, it is important to highlight what happens when you divide one integer by another integer in C++ because the result may not be what you expect. When both the dividend and divisor are integers, the quotient is always an integer in C++. For example, the result of the expression `24 / 5` is the integer 4 rather than the real number 4.8. So how do you get the quotient as a real number? You do so by converting at least one of the integers involved in the division operation to a real number. If one of the integers is a numeric literal constant, you can convert the literal constant to a real number by adding .0 to it. In this case, for example, you can change the expression `24 / 5` to `24.0 / 5`. When the computer evaluates the `24.0 / 5` expression, it will implicitly convert the integer 5 to the `double` number 5.0 before dividing it into the `double` number 24.0; the result will be the `double` number 4.8. You also can use either the expression `24 / 5.0` or the expression `24.0 / 5.0`; both expressions evaluate to 4.8. Similarly, if the `firstNum` variable contains the integer 5, the result of the expression `24.0 / firstNum` also is 4.8. This is because the integer stored in the `firstNum` variable will be implicitly promoted to the `double` data type before the division is performed. But what if neither of the integers involved in the division operation is a literal constant? For example, what if both the dividend and divisor are `int` variables? Now how do you get the quotient as a real number? In that case, you need to explicitly convert at least one of the `int` variables in the expression to either the `double` or `float` data type. (However, recall that the programs in this book will use the `double` data

type for real numbers.) You can use the `static_cast` operator to perform the conversion.

The static_cast Operator

C++ provides the **static_cast operator** for explicitly converting data from one data type to another. This type of conversion is called an **explicit type conversion** or a **type cast**. Figure 4-10 shows the `static_cast` operator's syntax and includes examples of using the operator. In the syntax, *data* can be a literal constant, named constant, or variable, and *dataType* is the data type to which you want the *data* converted. In the examples, `firstNum` and `secondNum` are `int` variables that contain the numbers 5 and 2, respectively. You can use any of the expressions shown in Examples 1 through 3 to divide the integer stored in the `firstNum` variable by the integer stored in the `secondNum` variable and then return the quotient as a real number having the `double` data type. The expression in Example 4 uses the `static_cast` operator to explicitly promote the integer stored in the `firstNum` variable to the `double` data type before it is multiplied by the `double` number 10.65. Although the same answer would be achieved with implicit type conversion, the type casting makes the programmer's intent clear to anyone reading the program. The statement in Example 5 uses the `static_cast` operator to explicitly convert, or type cast, the `double` number 3.99 to the `float` data type. In this case, the `double` number will be demoted to the `float` data type. (Recall that the `double` data type ranks higher than the `float` data type, because it can store larger numbers and with more precision.)

HOW TO Use the `static_cast` Operator

Syntax
static_cast<*dataType***>(***data***)**

Example 1 `static_cast<double>(firstNum) / static_cast<double>(secondNum)`
1. The value stored in the `firstNum` variable (the integer 5) is explicitly promoted to the `double` number 5.0.
2. The value stored in the `secondNum` variable (the integer 2) is explicitly promoted to the `double` number 2.0.
3. The `double` number 5.0 (the result of Step 1) is divided by the `double` number 2.0 (the result of Step 2). The result of the division is the `double` number 2.5.

Example 2 `static_cast<double>(firstNum) / secondNum`
1. The value stored in the `firstNum` variable (the integer 5) is explicitly promoted to the `double` number 5.0.
2. The value stored in the `secondNum` variable (the integer 2) is implicitly promoted to the `double` number 2.0.
3. The `double` number 5.0 (the result of Step 1) is divided by the `double` number 2.0 (the result of Step 2). The result of the division is the `double` number 2.5.

(continues)

Figure 4-10 How to use the `static_cast` operator

(continued)

Example 3 `firstNum / static_cast<double>(secondNum)`

1. The value stored in the `secondNum` variable (the integer 2) is explicitly promoted to the `double` number 2.0.
2. The value stored in the `firstNum` variable (the integer 5) is implicitly promoted to the `double` number 5.0.
3. The `double` number 5.0 (the result of Step 2) is divided by the `double` number 2.0 (the result of Step 1). The result of the division is the `double` number 2.5.

Example 4 `10.65 * static_cast<double>(firstNum)`
The value stored in the `firstNum` variable (the integer 5) is explicitly promoted to the `double` number 5.0 before being multiplied by the `double` number 10.65. The result is the `double` number 53.25. The `static_cast` operator is not required in this example, because the computer will implicitly convert the contents of the `firstNum` variable to the `double` data type before performing the multiplication operation.

Example 5 `const float PRICE = static_cast<float>(3.99);`
The `double` number 3.99 is explicitly converted to the `float` data type before being stored in the `PRICE` named constant.

Figure 4-10 How to use the `static_cast` operator

In most cases, the result of an arithmetic expression is assigned to a variable in a program. You do this using an assignment statement.

Assignment Statements

You can use an **assignment statement** to assign a value to a variable while a program is running. Figure 4-11 shows the syntax of an assignment statement in C++. The = symbol in the syntax is called the **assignment operator**. The figure also includes examples of assignment statements. (For clarity, the variable declaration statements are included in the examples.) An assignment statement tells the computer to evaluate the expression that appears on the right side of the assignment operator and then store the result in the variable whose name appears on the left side of the assignment operator. The expression can include one or more items, and the items can be literal constants, named constants, variables, or arithmetic operators. As with declaration statements, the data type of the expression in an assignment statement must match the data type of the variable to which the expression is assigned. As you learned in Chapter 3, when a value's data type does not match the memory location's data type, the computer uses a process called implicit type conversion to convert the value to fit the memory location. However, recall that implicit type conversions—more specifically, those that demote the value—do not always give you the expected results. Therefore, it is considered a good programming practice to use a type cast, if necessary, to explicitly convert the value of the expression to fit the memory location. When a value is assigned to a variable, it replaces the existing value in the memory location; this is because a variable can store only one value at any time.



You cannot use an assignment statement to assign a value to a named constant, because the contents of a named constant cannot be changed during runtime.

HOW TO Write an Assignment Statement

Syntax
variableName = *expression*;

Example 1
```
int quantity = 0;
quantity = 1000;
```
The assignment statement assigns the integer 1000 to the `quantity` variable.

Example 2
```
int janOrder = 500;
int febOrder = 225;
int total = 0;
total = janOrder + febOrder;
```
The assignment statement assigns the integer 725 to the `total` variable.

Example 3
```
int firstNum = 5;
int secondNum = 2;
double quotient = 0.0;
quotient = static_cast<double>(firstNum) / secondNum;
```
The assignment statement assigns the `double` number 2.5 to the `quotient` variable.

Example 4
```
char middleInitial = ' ';
middleInitial = 'C';
```
The assignment statement assigns the letter C to the `middleInitial` variable.

Example 5
```
string customerName = "";
customerName = "Jeff Brown";
```
The assignment statement assigns the string "Jeff Brown" to the `customerName` variable.

Figure 4-11 How to write an assignment statement

It is easy to confuse an assignment statement with a variable declaration statement in C++. For example, the assignment statement `hours = 50;` looks very similar to the variable declaration statement `int hours = 50;`. The noticeable difference is the data type that appears at the beginning of the declaration statement. However, keep in mind that a variable declaration statement creates (and optionally initializes) a *new* variable. An assignment statement, on the other hand, assigns a value to an *existing* variable.

Shaded in Figure 4-12 are the appropriate calculation statements for the Treyson Mobley problem. Because all of the items in both calculation statements have the same data type, neither statement requires any implicit or explicit type conversions.

IPO chart information	C++ instructions
Input	
total bill	`double totalBill = 0.0;`
liquor charge	`double liquor = 0.0;`
tip percentage	`double tipPercent = 0.0;`
Processing	
total bill without liquor charge	`double totalNoLiquor = 0.0;`
Output	
tip	`double tip = 0.0;`
Algorithm	

1. enter the total bill, liquor charge, and tip percentage	`cout << "Enter the total bill: ";`
	`cin >> totalBill;`
	`cout << "Enter the liquor charge: ";`
	`cin >> liquor;`
	`cout << "Enter the tip percentage in decimal format: ";`
	`cin >> tipPercent;`

2. calculate the total bill without liquor charge by subtracting the liquor charge from the total bill	`totalNoLiquor = totalBill - liquor;`
3. calculate the tip by multiplying the total bill without liquor charge by the tip percentage	`tip = totalNoLiquor * tipPercent;`
4. display the tip	`cout << "Tip: $" << tip << endl;`

Figure 4-12 Calculation statements for the Treyson Mobley problem

You now have coded the algorithm into a program, which is Step 4 in the problem-solving process. Before moving on to Step 5, it is important to caution you about a problem you might encounter when using real numbers in calculations. As mentioned in Chapter 3, not all real numbers can be represented exactly within the computer's internal memory. As a result, the answer to some calculations may not be accurate to the penny. For example, the expression `7.0 / 3.0` yields a quotient of 2.333333 . . . (with the number 3 repeating indefinitely). The number 2.333333... can be stored in the computer's memory only as an approximation. Because many real numbers cannot be stored precisely, some programmers do not use them in monetary calculations where accuracy to the penny is required. Instead, some programmers use integers, while others use special classes designed to perform precise arithmetic using real numbers. These special classes can be purchased from third-party vendors, such as Rogue Wave Software. You can learn more about the problem of using real numbers in monetary calculations by completing Computer Exercise 15 at the end of this chapter. The exercise also allows you to explore the use of integers in calculations. (Your instructor may require you to use integers in monetary calculations; however, for simplicity, this book will use real numbers.)



The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 4-2

1. Write a C++ assignment statement that multiplies the number 9.55 by the value stored in an `int` variable named `hours` and then assigns the result to a `double` variable named `grossPay`. Use implicit type conversion.

2. Rewrite the answer to Question 1 using a type cast (explicit type conversion).

3. In C++, the expression `7 / 2 * 4.5` will evaluate to _____, when it should evaluate to _____. Why does the expression evaluate incorrectly?

4. Rewrite the expression from Question 3 so that it will evaluate correctly.

Step 5—Desk-check the Program

The fifth step in the problem-solving process is to desk-check the program to make sure that each instruction in the algorithm was translated correctly. You should desk-check the program using the same sample data used to desk-check the algorithm; the results of both desk-checks should be the same. For your convenience when comparing the results of both desk-checks later in this section, Figure 4-13 shows the desk-check table that you completed for the Treyson Mobley algorithm in Chapter 2.

total bill	liquor charge	tip percentage	total bill without liquor charge	tip
~~45~~	~~10~~	~~.2~~	~~35~~	~~7~~
30	0	.15	30	4.50

Figure 4-13 Algorithm's desk-check table from Chapter 2

When desk-checking a program, you first place the names of the declared memory locations (variables and named constants) in a new desk-check table, along with each memory location's initial value. Figure 4-14 shows the result of desk-checking the variable declaration statements shown earlier in Figure 4-12.

totalBill	liquor	tipPercent	totalNoLiquor	tip
0.0	0.0	0.0	0.0	0.0

Figure 4-14 Variable names and initial values entered in the program's desk-check table

Next, you desk-check the remaining C++ instructions in order, recording in the desk-check table any changes made to the variables. In the Treyson Mobley program, the first instruction following the declaration statements

is the `cout << "Enter the total bill: ";` statement. The statement displays a prompt on the computer screen, but it does not make any changes to the program's variables; therefore, no entry is necessary in the desk-check table. The next statement, `cin >> totalBill;`, allows the user to enter the total bill amount, and it stores the user's response in the `totalBill` variable. If the user enters the number 45, the statement stores the number 45.0 in the variable, because the variable has the `double` data type. Therefore, you record 45.0 in the `totalBill` column in the desk-check table. (As you learned in Chapter 2, some programmers find it helpful to lightly cross out the previous value in a column before recording a new value; however, this is not a requirement.) Next, the `cout << "Enter the liquor charge: ";` statement prompts the user to enter the liquor charge. The `cin >> liquor;` statement waits for the user's response and then stores the response in the `liquor` variable. If the user enters the number 10, you record 10.0 in the desk-check table's `liquor` column. The `cout << "Enter the tip percentage in decimal format: ";` statement prompts the user to enter the tip percentage. The `cin >> tipPercent;` statement waits for the user's response and then stores the response in the `tipPercent` variable. If the user enters the number .2, you record .2 in the desk-check table. Figure 4-15 shows the input values recorded in the program's desk-check table.

totalBill	liquor	tipPercent	totalNoLiquor	tip
~~0.0~~	~~0.0~~	~~0.0~~	0.0	0.0
45.0	10.0	.2		

Figure 4-15 Input values entered in the program's desk-check table

The `totalNoLiquor = totalBill - liquor;` statement in the program subtracts the contents of the `liquor` variable (10.0) from the contents of the `totalBill` variable (45.0) and then stores the result (35.0) in the `totalNoLiquor` variable. Notice that the expression that appears on the right side of the assignment operator is evaluated first, and then the result is stored in the variable whose name appears on the left side of the assignment operator. As a result of this statement, you record 35.0 in the `totalNoLiquor` column in the desk-check table, as shown in Figure 4-16.

totalBill	liquor	tipPercent	totalNoLiquor	tip
~~0.0~~	~~0.0~~	~~0.0~~	~~0.0~~	0.0
45.0	10.0	.2	35.0	

Figure 4-16 Desk-check table showing the result of the total bill without liquor charge calculation

The next statement, `tip = totalNoLiquor * tipPercent;`, multiplies the contents of the `totalNoLiquor` variable (35.0) by the contents of the `tipPercent` variable (.2) and then stores the result (7.0) in the `tip` variable. In the desk-check table, you record the number 7.0 in the `tip` column, as shown in Figure 4-17.

totalBill	liquor	tipPercent	totalNoLiquor	tip
~~0.0~~	~~0.0~~	~~0.0~~	~~0.0~~	~~0.0~~
45.0	10.0	.2	35.0	7.0

Figure 4-17 Desk-check table showing the result of the tip calculation

The last statement in the program shown earlier in Figure 4-12 displays the `"Tip: $"` message along with the contents of the `tip` variable on the screen. You now have completed desk-checking the program using the first set of test data. If you compare the second row of values in Figure 4-17 with the first row of values shown earlier in Figure 4-13, you will notice that the results obtained when desk-checking the program are the same as the results obtained when desk-checking the algorithm. Recall, however, that you should perform several desk-checks (using different data) to make sure that the program works correctly. For the second desk-check, you will use 30, 0, and .15 as the total bill, liquor charge, and tip percentage, respectively. (This is the same data used in the second desk-check shown in Figure 4-13.) Each time you desk-check a program, keep in mind that you must complete all of the program's statements, beginning with the first statement and ending with the last statement. In this case, the first statement declares and initializes the `totalBill` variable, and the last statement displays the tip amount on the screen. The completed desk-check table is shown in Figure 4-18. Here again, if you compare the fourth row of values in Figure 4-18 with the second row of values shown earlier in Figure 4-13, you will notice that the program's results are the same as the algorithm's results.

totalBill	liquor	tipPercent	totalNoLiquor	tip
~~0.0~~	~~0.0~~	~~0.0~~	~~0.0~~	~~0.0~~
~~45.0~~	~~10.0~~	~~.2~~	~~35.0~~	~~7.0~~
~~0.0~~	~~0.0~~	~~0.0~~	~~0.0~~	~~0.0~~
30.0	0.0	.15	30.0	4.50

Figure 4-18 Program's desk-check table showing the results of the second desk-check

Step 6—Evaluate and Modify the Program

The final step in the problem-solving process is to evaluate and modify (if necessary) the program. You evaluate a program by entering your C++ instructions (along with other instructions that you will learn about later in this section) into the computer and then using the computer to run (execute) the program. While the program is running, you enter the same sample data used when desk-checking the program. If the results obtained when running the program differ from those shown in the program's desk-check table, it indicates that the program contains errors, referred to as **bugs**. The bugs must be located and removed from the program before the program is released to the user. The programmer's job is not finished until the program runs without errors and produces the expected results.

The process of locating and correcting the bugs in a program is called **debugging**. Program bugs typically are caused by either syntax errors or logic errors. A **syntax error** occurs when you break one of the programming language's rules. As you learned in Chapter 3, every programming language has a set of rules, called syntax, that you must follow when using the language. Most syntax errors are a result of typing errors that occur when entering instructions, such as typing `ednl` (instead of `endl`) or neglecting to enter a semicolon at the end of a statement. In most cases, syntax errors are easy to both locate and correct because they trigger an error message from the C++ compiler. The error message indicates the general vicinity of the error and includes a brief description of the error. Unlike syntax errors, logic errors are much more difficult to find, because they do not trigger an error message from the compiler. A **logic error** can occur for a variety of reasons, such as forgetting to enter an instruction or entering the instructions in the wrong order. Some logic errors occur as a result of calculation statements that are correct syntactically but incorrect mathematically. For example, consider the statement `average = number1 + number2 / 2;`, which is supposed to calculate the average of two numbers. The statement's syntax is correct, but it is incorrect mathematically. This is because it tells the computer to divide the contents of the `number2` variable by 2 and then add the quotient to the contents of the `number1` variable. (Recall that division is performed before addition in an arithmetic expression.) The correct instruction for calculating the average of two numbers is `average = (number1 + number2) / 2;`. The parentheses tell the computer to add the contents of the `number1` variable to the contents of the `number2` variable before dividing the sum by 2.

In order to enter your C++ instructions into the computer, you need to have access to a text editor, more simply referred to as an editor. The instructions you enter are called **source code**. You save the source code in a file on a disk, giving it the filename extension .cpp (which stands for *C plus plus*). The .cpp file is called the **source file**. In order to run (execute) the code contained in the source file, you need a C++ compiler. As you learned in Chapter 1, a compiler translates high-level instructions into machine code—the 0s and 1s that the computer can understand. Machine code is usually called **object code**. The compiler generates the object code and saves it in a file whose filename extension is .obj (which stands for *object*). The file containing the object code is called the **object file**. After the compiler creates the object file, it then invokes another program called a linker. The **linker** combines the object file with other machine code necessary for your C++ program to run correctly, such as machine code that allows the program to communicate with input and output devices. The linker produces an **executable file**, which is a file that contains all of the machine code necessary to run your C++ program as many times as desired without the need for translating the program again. The executable file has an extension of .exe on its filename. (The exe stands for *executable*.) Figure 4-19 illustrates the sequence of steps followed when translating your source code into executable code.

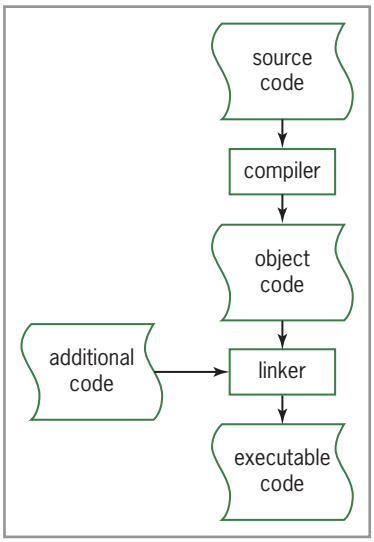


Figure 4-19 Process by which source code is translated into executable code

Many C++ development tools contain both the editor and compiler in one integrated environment, referred to as an **IDE** (**Integrated Development Environment**). Examples include Microsoft Visual C++, Borland C++ Builder, and Dev C++. Other C++ development tools, called command-line compilers, contain only the compiler and require you to use a general-purpose editor (such as Notepad, WordPad, or vi) to enter the program instructions into the computer. Appendix D in this book contains instructions for entering and running programs using the Microsoft Visual C++ 2010 IDE. Appendix E contains the instructions for using the IDE in Dev C++. However, keep in mind that you can enter and run the programs in this book using most C++ development tools, often with little or no modification.



You do not have to align the initial values in declaration statements as shown in Figure 4-20. However, doing so makes it easier to verify that each memory location has been initialized.

Figure 4-20 shows the source code for the Treyson Mobley program. Each line in the figure is numbered so that it is easier to refer to it in the text; you do not enter the line numbers in the program. The unshaded lines of code are your C++ instructions from Figure 4-12. Besides entering your C++ instructions, you also need to enter other instructions in the source file. Some of the additional instructions are required by the C++ compiler, while others are optional but highly recommended. The additional instructions are shaded in Figure 4-20.

```
 1 //Fig4-20.cpp - displays the amount of a tip
 2 //Created/revised by <your name> on <current date>
 3
 4 #include <iostream>
 5 using namespace std;
 6
 7 int main()
 8 {
 9     //declare variables
10     double totalBill     = 0.0;
```

function header

Figure 4-20 Treyson Mobley program *(continues)*

(continued)

```
11      double liquor        = 0.0;
12      double tipPercent    = 0.0;
13      double totalNoLiquor = 0.0;
14      double tip           = 0.0;
15
16      //enter input items
17      cout << "Enter the total bill: ";
18      cin >> totalBill;
19      cout << "Enter the liquor charge: ";
20      cin >> liquor;
21      cout << "Enter the tip percentage in decimal format: ";
22      cin >> tipPercent;
23
24      //calculate the total without liquor and the tip
25      totalNoLiquor = totalBill - liquor;
26      tip = totalNoLiquor * tipPercent;
27
28      //display the output item
29      cout << "Tip: $" << tip << endl;
30
31      system("pause");
32      return 0;
33  }   //end of main function
```

depending on your C++ development tool, this statement may not be necessary

Figure 4-20 Treyson Mobley program



Everything after the two forward slashes (//) to the end of the line is treated as a comment.

Lines 1, 2, 9, 16, 24, and 28 in the program are comments; there also is a comment on line 33. You create a comment by typing two forward slashes (//) before the text you want treated as a comment. A **comment** is simply a message to the person reading the program and is referred to as internal documentation. The comments on lines 1 and 2 indicate the program's name and purpose, as well as the programmer's name and the date the program was either created or revised. The remaining comments explain various sections of the code. The comment on line 9, for example, indicates that the instructions that follow it are variable declaration statements. The C++ compiler does not require you to include comments in a program. However, it is a good programming practice to do so because they make your code more readable and easier to understand by anyone viewing it. The C++ compiler does not process (execute) the comments in a program. Instead, the compiler ignores the comments when it translates the source code into object code. Comments do not end with a semicolon, because they are not statements in C++.

Lines 4 and 5 in Figure 4-20 are directives; line 4 is a #include directive and line 5 is a using directive. A **#include directive** provides a convenient way to merge the source code from one file with the source code in another file, without having to retype the code. The `#include <iostream>` directive, for example, tells the C++ compiler to include the contents of the iostream file in the current program. The iostream file must be included in any program that uses the `cin` or `cout` objects. A #include directive is not a C++ statement; therefore, it does not end with a semicolon. A using directive, on the other hand, *is* a statement; therefore, it must end with a semicolon. A **using directive** tells the compiler where (in the computer's internal memory) it can find the definitions

C++ programs typically contain at least one directive, and most contain many directives.

of keywords and classes, such as `double` or `string`. The `using namespace std;` directive indicates that the definitions of the standard C++ keywords and classes are located in the `std` (which stands for *standard*) namespace. A namespace is a special area in the computer's internal memory.

In line 7 of the program, `main` is the name of a function and must be typed using lowercase letters. A **function** is a block of code that performs a task. Functions have parentheses following their names, like this: `main()`. Some functions require you to enter information between the parentheses; other functions, like `main`, do not. Every C++ program must have a `main` function, because that is where the execution of a C++ program always begins. A C++ program can contain many functions; however, only one can be the `main` function. Some functions, like `main`, return a value after completing their assigned task. If a function returns a value, the data type of the value it returns appears to the left of the function name; otherwise, the keyword `void` appears to the left of the name. The `int` in line 7 indicates that the `main` function returns an integer. The entire line of code, `int main()`, is referred to as a **function header**, because it marks the beginning of the function. After the function header, you enter the code that directs the function on how to perform its assigned task. Examples of such code include statements that declare variables, as well as statements that input, calculate, and output data. In C++, you enclose a function's code within a set of braces ({}). The braces mark the beginning and end of the code block that comprises the function. You enter the opening brace ({) immediately below the function's header in the program, and you enter the closing brace (}) at the end of the function. In Figure 4-20, the opening brace appears on line 8, immediately below the `int main()` function header; the closing brace appears on line 33. Everything between the opening and closing braces in Figure 4-20 is included in the `main` function and is referred to as the **function body**. Notice that you can include a comment (in this case, `//end of main function`) on the same line with a C++ instruction. However, you must be sure to enter the comment *after* the instruction, because only the text appearing after the `//` on a line is interpreted as a comment.

Some C++ development tools automatically pause program execution and display the *Press any key to continue* message when a program ends, so they do not require the `system("pause");` statement.

Lines 31 and 32 in Figure 4-20 contain the `system("pause");` and `return 0;` statements. The `system("pause");` statement, whose requirement depends on the C++ development tool you are using, pauses program execution and displays the *Press any key to continue* message in a Command Prompt window on the computer screen, as shown in Figure 4-21. The `return 0;` statement returns the number 0 to the operating system to indicate that the program ended normally. (As mentioned earlier, the `main` function returns an integer.)

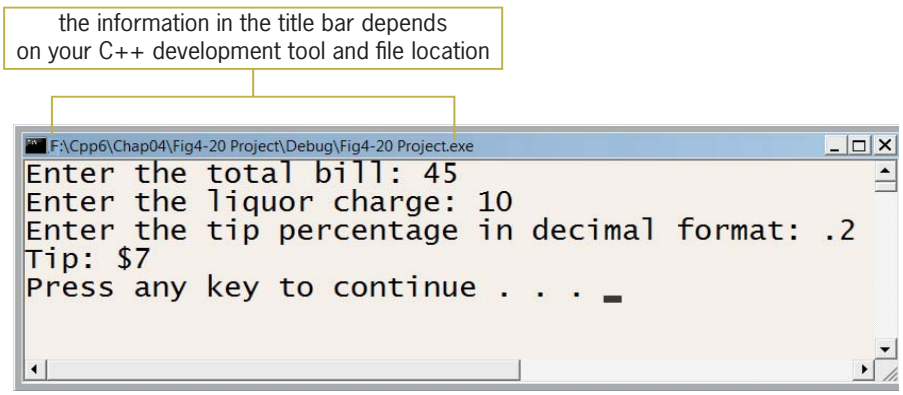

the information in the title bar depends on your C++ development tool and file location
F:\Cpp6\Chap04\Fig4-20 Project\Debug\Fig4-20 Project.exe
Enter the total bill: 45
Enter the liquor charge: 10
Enter the tip percentage in decimal format: .2
Tip: $7
Press any key to continue . . .

Figure 4-21 Command Prompt window

Arithmetic Assignment Operators

In addition to the standard arithmetic operators listed earlier in Figure 4-7, C++ also provides several arithmetic assignment operators. The **arithmetic assignment operators** allow you to abbreviate an assignment statement that contains an arithmetic operator. However, the assignment statement must have the following format, in which *variableName* is the name of the same variable: *variableName = variableName arithmeticOperator value*. For example, you can use the multiplication assignment operator (*=) to abbreviate the statement `price = price * 1.05;` as follows: `price *= 1.05;`. Both statements tell the computer to multiply the contents of the `price` variable by 1.05 and then store the result in the `price` variable. Figure 4-22 shows the syntax of a C++ statement that uses an arithmetic assignment operator. The figure also lists the most commonly used arithmetic assignment operators, and it includes examples of using an arithmetic assignment operator to abbreviate an assignment statement. Notice that each arithmetic assignment operator consists of an arithmetic operator followed immediately by the assignment operator (=). The arithmetic assignment operators do not contain a space; in other words, the addition assignment operator is +=, not + =. Including a space in an arithmetic assignment operator is a common syntax error.



In most cases, *value* is either a constant (literal or named) or the name of a different variable.

HOW TO Use an Arithmetic Assignment Operator

Syntax
variableName arithmeticAssignmentOperator value;

Operator	Purpose
+=	addition assignment
-=	subtraction assignment
*=	multiplication assignment
/=	division assignment
%=	modulus assignment

Example 1
Original statement: `rate = rate + .05;`
Abbreviated statement: `rate += .05;`

Example 2
Original statement: `price = price - discount;`
Abbreviated statement: `price -= discount;`



It's easy to abbreviate an assignment statement. Simply remove the variable name that appears on the left side of the assignment operator (=) in the statement, and then put the assignment operator immediately after the arithmetic operator.

Figure 4-22 How to use an arithmetic assignment operator

Mini-Quiz 4-3



The answers to Mini-Quiz questions are located in Appendix A.

1. Typing `cin > age;` rather than `cin >> age;` is an example of a _____ error.

2. The .cpp file that contains your C++ instructions is called the _____ file.
   a. executable
   b. object
   c. source
   d. statement

3. In a C++ program, the body of a function is enclosed in _____.
   a. braces
   b. parentheses
   c. square brackets
   d. none of the above

4. Rewrite the `age = age + 1;` statement using an arithmetic assignment operator.



The answers to the labs are located in Appendix A.



LAB 4-1 Stop and Analyze

Study the three examples shown in Figure 4-23, and then answer the questions.

Example 1

```
int numberOfPeople = 10;
double costPerPerson = 7.45;
double totalCost = 0.0;
totalCost = numberOfPeople * costPerPerson;
numberOfPeople = numberOfPeople / 2;
costPerPerson = costPerPerson + 3;
```

Example 2

```
double score1 = 100.0;
double score2 = 90.0;
double average = 0.0;
average = score1 + score2 / 2;
```

Example 3

```
int juneSales = 933;
int julySales = 1216;
double avgSales = 0.0;
avgSales = (juneSales + julySales) / 2;
```

Figure 4-23 Examples for Lab 4-1

QUESTIONS

1. Explain how the computer evaluates the total cost calculation statement in Example 1. What value will be assigned to the `totalCost` variable? Is the value correct? If not, how can you fix the statement so it evaluates correctly?

2. Explain how the computer evaluates the number of people calculation statement in Example 1. The statement should divide the number of people by 2 and then assign the result to the `numberOfPeople` variable. What value will be assigned to the variable? Is the value correct? If not, how can you fix the statement so it evaluates correctly?
3. Explain how the computer evaluates the cost per person calculation statement in Example 1. What value will be assigned to the `costPerPerson` variable? Is the value correct? If not, how can you fix the statement so it evaluates correctly?
4. Explain how the computer evaluates the average calculation statement in Example 2. What value will be assigned to the `average` variable? Is the value correct? If not, how can you fix the statement so it evaluates correctly?
5. Explain how the computer evaluates the average sales calculation statement in Example 3. What value will be assigned to the `avgSales` variable? Is the value correct? If not, how can you fix the statement so it evaluates correctly?



LAB 4-2 Plan and Create

In this lab, you will plan and create an algorithm that displays the total amount a student owes for a semester. The problem specification is shown in Figure 4-24.

> The cashier at Hoover College wants a program that calculates and displays the total amount a student owes for the semester, including tuition and room and board. The fee per semester hour is $100, and room and board is $2000 per semester. Courses at the college can be 1, 2, or 3 semester hours.

Figure 4-24 Problem specification for Lab 4-2

First, analyze the problem, looking for the output first and then for the input. In this case, the user wants the program to display the total amount the student owes. To calculate the total amount, the computer will need to know the number of semester hours for which the student is enrolled, as well as the fee per semester hour and the room and board fee. The number of semester hours will be entered by the user, whereas the problem specification indicates that the fee per semester hour is $100, and the room and board fee is $2000. Figure 4-25 shows the input and output items entered in an IPO chart.

Input	Processing	Output
hours enrolled fee per hour (100) room & board fee (2000)	Processing items: Algorithm:	total owed

Figure 4-25 Partially completed IPO chart showing the input and output items

As you know, most algorithms begin with an instruction to enter the input items into the computer, followed by instructions that process the input items, typically including the items in one or more calculations. Most algorithms end with one or more instructions that display, print, or store the output items. Figure 4-26 shows the completed IPO chart for the Hoover College problem.

Input	Processing	Output
hours enrolled fee per hour (100) room & board fee (2000)	Processing items: none Algorithm: 1. enter the hours enrolled 2. calculate the total owed by multiplying the hours enrolled by the fee per hour and then adding the room & board fee to the result 3. display the total owed	total owed

Figure 4-26 Completed IPO chart for the Hoover College problem

After completing the IPO chart, you then move on to the third step in the problem-solving process, which is to desk-check the algorithm. You will desk-check the Hoover College algorithm twice, using 9 and 11 as the number of hours enrolled. Manually calculating the total owed results in $2900 for the first desk-check and $3100 for the second desk-check. Figure 4-27 shows the completed desk-check table. Notice that the amounts in the total owed column agree with the results of the manual calculations.

hours enrolled	fee per hour	room & board fee	total owed
~~9~~	~~100~~	~~2000~~	~~2900~~
11	100	2000	3100

Figure 4-27 Completed desk-check table for the Hoover College algorithm

The fourth step in the problem-solving process is to code the algorithm into a program. You begin by declaring memory locations that will store the values of the input, processing (if any), and output items. The Hoover College problem will need four memory locations to store the values of the hours enrolled, fee per hour, room & board fee, and total owed. The hours enrolled and total owed values should be stored in variables, because the user should be allowed to change the hours enrolled value, which then will change the total owed value, while the program is running. The fee per hour and room & board fee, however, will be stored in named constants, because those values should not change during runtime. The variables and named constants will store integers, so you will use the `int` data type for each one. Figure 4-28 shows the input, processing, and output items from the IPO chart, along with the corresponding C++ statements.

IPO chart information	C++ instructions
Input	
hours enrolled	`int hours = 0;`
fee per hour (100)	`const int FEE_PER_HOUR = 100;`
room & board fee (2000)	`const int ROOM_BOARD = 2000;`
Processing	
none	
Output	
total owed	`int totalOwed = 0;`
Algorithm	
1. enter the hours enrolled	`cout << "Hours enrolled? ";` `cin >> hours;`
2. calculate the total owed by multiplying the hours enrolled by the fee per hour and then adding the room & board fee to the result	`totalOwed = hours * FEE_PER_HOUR + ROOM_BOARD;`
3. display the total owed	`cout << "Total owed: $" << totalOwed << endl;`

Figure 4-28 IPO chart information and C++ instructions for the Hoover College problem

The fifth step in the problem-solving process is to desk-check the program. You begin by placing the names of the declared variables and named constants in a new desk-check table, along with their values. You then desk-check the remaining C++ instructions in order, recording in the desk-check table any changes made to the variables. Figure 4-29 shows the completed desk-check table for the program. The results agree with those shown in the algorithm's desk-check table in Figure 4-27.

hours	FEE_PER_HOUR	ROOM_BOARD	totalOwed	
~~0~~	~~100~~	~~2000~~	~~0~~	first desk-check
~~9~~			~~2900~~	
~~0~~	100	2000	~~0~~	second desk-check
11			3100	

Figure 4-29 Completed desk-check table for the Hoover College program

The final step in the problem-solving process is to evaluate and modify (if necessary) the program. Recall that you evaluate a program by entering its instructions into the computer and then using the computer to run (execute) it. While the program is running, you enter the same sample data used when desk-checking the program. The method of entering the instructions and running the program depends on the C++ IDE you are using; or, if you are not using an IDE, it depends on your compiler and text editor. As mentioned earlier, Appendices D and E show you how to enter and run programs using Microsoft Visual C++ 2010 and Dev C++, respectively. However, you can enter and run the programs in this book using most C++ systems, often with

little or no modification. Your instructor or technical support person will provide you with the appropriate instructions if you are not using Microsoft Visual C++ 2010 or Dev C++.

DIRECTIONS

Follow the instructions for starting your C++ development tool. Depending on the development tool you are using, you may need to create a new project; if so, name the project Lab4-2 Project and save it in the Cpp6\Chap04 folder. Enter the instructions shown in Figure 4-30 in a source file named Lab4-2.cpp. (Do not enter the line numbers.) Save the file in either the project folder or the Cpp6\Chap04 folder. Now follow the appropriate instructions for running the Lab4-2.cpp file. Run the program twice, using the sample data values of 9 and 11 for the hours enrolled. If necessary, correct any bugs (errors) in the program.

```
 1 //Lab4-2.cpp - displays the total owed
 2 //Created/revised by <your name> on <current date>
 3
 4 #include <iostream>
 5 using namespace std;
 6
 7 int main()
 8 {
 9    //declare variables and named constants
10    int hours              = 0;
11    int totalOwed          = 0;
12    const int FEE_PER_HOUR = 100;
13    const int ROOM_BOARD   = 2000;
14
15    //enter hours enrolled
16    cout << "Hours enrolled? ";
17    cin >> hours;
18
19    //calculate total owed
20    totalOwed = hours * FEE_PER_HOUR + ROOM_BOARD;
21
22    //display total owed
23    cout << "Total owed: $" << totalOwed << endl;
24
25    system("pause");
26    return 0;
27 }  //end of main function
```

if your C++ development tool does not require this statement, either omit it or make it a comment

Figure 4-30 Hoover College program



LAB 4-3 Modify

If necessary, create a new project named Lab4-3 Project. Enter (or copy) the Lab4-2.cpp instructions into a new source file named Lab4-3.cpp. Hoover College now has courses that can be .5, 1, 2,

or 3 semester hours. In addition, the fee per hour has been raised to $105. Modify the program instructions. Be sure to change Lab4-2.cpp in the first comment to Lab4-3.cpp. Test the program twice, using 9.5 and 11 as the number of hours enrolled. The total owed should be $2997.5 and $3155. (Don't be concerned that the $2997.5 has only one decimal place. You will learn how to format numbers in Chapter 5.)



LAB 4-4 Desk-Check

Desk-check the three lines of code shown in Figure 4-31.

```
int num 75;
int answer = 0;
answer = num % 2;
```

Figure 4-31 Code for Lab 4-4



LAB 4-5 Debug

Follow the instructions for starting C++ and opening the Lab4-5.cpp file. If necessary, make the `system("pause");` statement a comment, and then save the program. Run and then debug the program.

Summary

- The fourth step in the problem-solving process is to code the algorithm. You begin by declaring a memory location for each unique input, processing, and output item listed in the IPO chart. You then translate each instruction in the algorithm into one or more C++ statements.
- In C++, you perform standard input and output operations using stream objects. The standard input stream object is called `cin`. The standard output stream object is called `cout`.
- You use `cin` along with the extraction operator (>>) to get either numeric or character input from the computer keyboard. You use `cout` along with the insertion operator (<<) to display information on the computer screen.
- The extraction operator stops removing characters from the `cin` object when it encounters a white-space character.
- The `endl` stream manipulator advances the cursor to the next line on the computer screen.

- A program should display (on the computer screen) a separate and meaningful prompt for each item of data the user should enter.
- You direct the computer to perform a calculation by writing an arithmetic expression that contains one or more arithmetic operators.
- Each arithmetic operator is associated with a precedence number, which controls the order in which the operation is performed in an expression. When an arithmetic expression contains more than one operator having the same priority, those operators are evaluated from left to right. You can use parentheses to override the normal order of precedence.
- When an arithmetic operation involves two values having different data types, the computer implicitly promotes the value with the lower-ranking data type to the higher-ranking data type. The value returns to its original data type upon completion of the arithmetic operation.
- The quotient obtained by dividing one integer by another integer is always an integer in C++.
- You can use the `static_cast` operator to explicitly convert data from one data type to another.
- You can use an assignment statement to assign a value to a variable during runtime. An assignment statement tells the computer to evaluate the expression that appears on the right side of the assignment operator (=) and then store the result in the variable whose name appears on the left side of the assignment operator.
- The fifth step in the problem-solving process is to desk-check the program. You should use the same sample data used to desk-check the algorithm.
- The sixth (and final) step in the problem-solving process is to evaluate and modify (if necessary) the program.
- The errors in a program are called bugs and typically fall into one of two categories: syntax errors or logic errors.
- In order for you to enter your C++ instructions into the computer and then run the program, you need to have access to a text editor and a C++ compiler.
- The C++ instructions entered in a program are called source code and are saved in a source file, which has a .cpp filename extension.
- The compiler translates source code into machine code, also called object code.
- The linker produces an executable file that contains all the machine code necessary to run a C++ program. The executable file has an .exe filename extension.
- Programmers use comments to document a program internally. Doing this makes the program easier to understand by anyone viewing it. Comments are not statements and are ignored by the compiler.
- The `#include <iostream>` directive tells the computer to include the contents of the iostream file in the current program.
- The `using namespace std;` directive tells the computer that the definitions of standard C++ keywords and classes are located in the `std` namespace. A namespace is a special area in the computer's internal memory.

- The execution of a C++ program begins with the `main` function. Therefore, every C++ program must have one (and only one) `main` function.
- The first line in a function is called the function header. Following the function header is the function body, which must be enclosed in braces.
- C++ provides arithmetic assignment operators that allow you to abbreviate an assignment statement as follows: *variableName arithmeticAssignmentOperator value*;. However, the original assignment statement must have the following format, in which *variableName* is the name of the same variable: *variableName* = *variableName arithmeticOperator value*.

Key Terms

`#include` directive—an instruction that tells the computer to merge the source code from one file with the source code from another file

%—modulus operator; divides two integers and returns the remainder as an integer

<<—the insertion operator in C++

>>—the extraction operator in C++

Arithmetic assignment operators—operators comprised of an arithmetic operator and the assignment operator; allow you to abbreviate an assignment statement that follows a specific format

Assignment operator—the = symbol in an assignment statement

Assignment statement—used to assign a value to a variable during runtime

Bugs—the errors in a program

`cin`—the standard input stream object in C++; tells the computer to pause program execution while the user enters one or more characters at the keyboard; temporarily stores the characters entered at the keyboard

Comment—a message used to document a program internally; begins with two forward slashes (//) in C++

`cout`—the standard output stream object in C++; used with the insertion operator to display information on the computer screen

Debugging—the process of locating and correcting any errors in a program

`endl`—a stream manipulator that can be used to advance the cursor to the next line on the computer screen

Executable file—a file that contains all of the machine code necessary to run a program; executable files have an .exe filename extension

Explicit type conversion—the explicit conversion of data from one data type to another; usually performed with the `static_cast` operator; also called a type cast

Extraction operator—two greater-than signs (>>); extracts (removes) characters from the `cin` object and sends them "in" to the computer's internal memory

Function—a block of code that performs a task

Function body—the code contained between a function's opening and closing braces

Function header—the first line in a function; marks the beginning of the function

IDE—an acronym for Integrated Development Environment

Insertion operator—two less-than signs (<<); used with the `cout` object to display information on the computer screen

Integrated Development Environment—a system that includes both an editor and a compiler

Linker—a program that combines the code contained in a C++ program's object file with other machine code necessary to run the C++ program

Logic error—an error (bug) that occurs when you neglect to enter a program instruction or enter the instructions in the wrong order; also occurs as a result of calculation statements that are correct syntactically but incorrect mathematically

Modulus operator—the percent sign (%); divides two integers and returns the remainder as an integer

Object code—another name for machine code

Object file—a file that contains the object code associated with a program; automatically generated by the compiler

Prompt—a message (displayed on the computer screen) indicating the type of data the user should enter at the keyboard

Source code—the program instructions you enter using an editor; the instructions are saved in a source file

Source file—a file that contains a program's source code; source files have a .cpp filename extension

`static_cast` operator—explicitly converts (or type casts) data from one data type to another

Stream—a sequence of characters

Stream manipulator—allows a C++ program to manipulate (or manage) the characters in either the input or output stream

Stream objects—objects used to perform standard input and output operations in C++

Syntax error—an error (bug) that occurs when a program instruction violates a programming language's syntax

Testing—running (executing) a program, along with sample data, on the computer

Type cast—another term for an explicit type conversion

`using` directive—an instruction that tells the computer where it can find the definitions of keywords and classes

White-space character—a newline character, a tab character, or a blank (space) character

Review Questions

1. Which of the following prompts the user to enter a price?

   a. `cin >> "What is the price? ";`

   b. `cin << "What is the price? ";`

   c. `cout << "What is the price? ";`

   d. `cout >> "What is the price? ";`

2. Which of the following sends keyboard input to a variable named `price`?

   a. `cin >> price;`

   b. `cin << price;`

   c. `cin <> price;`

   d. `cin > price;`

3. If the `price` variable has the `double` data type, which of the following statements will require an explicit type conversion to evaluate correctly?

   a. `price = 25;`

   b. `price = price * 1.05;`

   c. `price = price / 2;`

   d. none of the above

4. The `num1` and `num2` variables have the `int` data type and contain the numbers 13 and 5, respectively. The `answer` variable has the `double` data type. Which of the following statements will require an explicit type conversion to evaluate correctly?

   a. `answer = num1 / 4.0;`

   b. `answer = num1 + num1 / num2;`

   c. `answer = num1 - num2;`

   d. none of the above

5. Which of the following assigns the letter T to a `char` variable named `insured`?

   a. `insured = 'T';`

   b. `insured = "T";`

   c. `insured = T;`

   d. none of the above

6. Which of the following explicitly converts the contents of an `int` variable named `quantity` to the `double` data type?

   a. `castToDouble(quantity);`

   b. `explicit_cast<double>(quantity);`

   c. `static_cast<double>(quantity);`

   d. `type_cast<double>(quantity);`

7. Which of the following statements advances the cursor to the next line on the computer screen?

   a. `cout << endl;`

   b. `cout << endline;`

   c. `cout << newline;`

   d. none of the above

8. Which of the following tells the compiler to merge the code contained in the iostream file with the current file's code?

   a. `#include iostream;`

   b. `#include <iostream>`

   c. `#include <iostream>;`

   d. `#include (iostream)`

9. Which of the following is equivalent to the `rate = rate / 100;` statement?

   a. `rate = rate =/ 100;`

   b. `rate /= 100;`

   c. `rate =/ 100;`

   d. none of the above

10. Which of the following is a valid comment in C++?

    a. `**This is a comment`

    b. `@/This is a comment`

    c. `/This is a comment`

    d. none of the above

Exercises



Pencil and Paper

TRY THIS

1. Complete the C++ instructions column in Figure 4-32. (The answers to TRY THIS Exercises are located at the end of the chapter.)

IPO chart information	C++ instructions
Input	
sale1	`double sale1 = 0.0;`
sale2	`double sale2 = 0.0;`
commission rate	`double commissionRate = 0.0;`
Processing	
total sales	`double totalSales = 0.0;`
Output	
commission	`double commission = 0.0;`
Algorithm	
1. enter sale1, sale2, and the commission rate	
2. calculate the total sales by adding sale1 to sale2	
3. calculate the commission by multiplying the total sales by the commission rate	
4. display the commission	

Figure 4-32

TRY THIS

2. Complete the C++ instructions column in Figure 4-33. (The answers to TRY THIS Exercises are located at the end of the chapter.)

IPO chart information	C++ instructions
Input	
cup price	`double cupPrice = 0.0;`
plate price	`double platePrice = 0.0;`
cups purchased	`int cupsPurchased = 0;`
plates purchased	`int platesPurchased = 0;`
sales tax rate (5.5%)	`const double TAX_RATE = .055;`
Processing	
total cup cost	`double totalCupCost = 0.0;`
total plate cost	`double totalPlateCost = 0.0;`
subtotal	`double subtotal = 0.0;`
Output	
total cost	`double totalCost = 0.0;`

Figure 4-33 *(continues)*

(continued)

Algorithm

1. enter cup price, plate price, cups purchased, and plates purchased
2. calculate the total cup cost by multiplying the cups purchased by the cup price
3. calculate the total plate cost by multiplying the plates purchased by the plate price
4. calculate the subtotal by adding the total cup cost to the total plate cost
5. calculate the total cost by multiplying the subtotal by the sales tax rate and then adding the result to the subtotal
6. display the total cost

Figure 4-33

MODIFY THIS

3. Complete TRY THIS Exercise 1, and then modify the IPO chart information and C++ instructions so that the commission rate will always be 10%.

INTRODUCTORY

4. Complete the C++ instructions column in Figure 4-34.

IPO chart information	C++ instructions
Input	
first number	`double num1 = 0.0;`
second number	`double num2 = 0.0;`
Processing	
none	
Output	
quotient	`double quotient = 0.0;`
Algorithm	
1. enter the first number and second number	`cout << "First number: ";` `cin >> num1;`
	_____
	_____
2. calculate the quotient by dividing the first number by the second number	_____
3. display the quotient	`cout << "The quotient is "` `<< quotient << endl;`

Figure 4-34

5. Complete the C++ instructions column in Figure 4-35. INTERMEDIATE

IPO chart information	C++ instructions
Input	
miles driven	`int milesDriven = 0;`
gallons used	`int gallonsUsed = 0;`
Processing	
none	
Output	
miles per gallon	`double milesPerGal = 0.0;`
Algorithm	
1. enter the miles driven and gallons used	_____
	_____
	_____
	_____
2. calculate the miles per gallon by dividing the miles driven by the gallons used	_____
3. display the miles per gallon	`cout << "MPG: " << milesPerGal << endl;`

Figure 4-35

6. Complete the C++ instructions column in Figure 4-36. ADVANCED

IPO chart information	C++ instructions
Input	
assessed value	`int assessedValue = 0;`
tax rate ($1.02)	_____
Processing	
none	
Output	
annual property tax	`double tax = 0.0;`
Algorithm	
1. enter the assessed value	_____
	_____
2. calculate the annual property tax by dividing the assessed value by 100 and then multiplying the result by the tax rate	_____
3. display the annual property tax	_____

Figure 4-36

SWAT THE BUGS

7. Correct the errors in the lines of code shown in Figure 4-37. (The code contains nine errors.)

```
#include <iostreem>
using std namespace;

int main
{
      /declare variables
      Int quantity = 0;

      //enter the input item
      cout "Enter the quantity ordered: ";
      cin << quantity;

      //display a message
      cout << "You entered " << quantity << endl

      system("pause");
      return 0;
{     //end of main function
```

Figure 4-37



Computer

TRY THIS

8. The answer to TRY THIS Exercise 1 is shown in Figure 4-38 at the end of the chapter. Enter the C++ instructions from the figure into a source file named TryThis8.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Save and run the program. Test the program using $345.55 and $576.34 as the two sale amounts and .05 as the commission rate. The answer should be $46.0945. (Don't be concerned if your answer has more decimal places.) Now test it using $3000, $2500, and .06. (The answers to TRY THIS Exercises are located at the end of the chapter.)

TRY THIS

9. The answer to TRY THIS Exercise 2 is shown in Figure 4-39 at the end of the chapter. Enter the C++ instructions from the figure into a source file named TryThis9.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Save and run the program. Test the program using $.50 as the cup price, $1.05 as the plate price, 35 as the number of cups purchased, and 35 as the number of plates purchased. The answer should be $57.2338. (Don't be concerned if your answer has more decimal places.) Then test it using $.25, $.75, 20, and 10. (The answers to TRY THIS Exercises are located at the end of the chapter.)

MODIFY THIS

10. Complete TRY THIS Exercise 8. Enter (or copy) the instructions from the TryThis8.cpp file into a new source file named ModifyThis10.cpp. Modify the code in the ModifyThis10.cpp file so that the commission rate will always be 10%. Save and run the program. Test the program using $345.55 and $576.34 as the two sale amounts. Then test it using $3000 and $2500.

11. A-1 Appliances needs a program that allows the store clerks to enter the number of dishwashers in stock at the beginning of the month, the number purchased during the month, and the number sold during the month. The program should calculate and display the number of dishwashers in stock at the end of the month. INTRODUCTORY

    a. Create an IPO chart for the problem, and then desk-check the algorithm twice. For the first desk-check, use 5000 as the number of dishwashers at the beginning of the month, 1000 as the number purchased, and 3500 as the number sold. For the second desk-check, use 450, 20, and 125.

    b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown in Figure 4-12 in the chapter. Then code the algorithm into a program.

    c. Desk-check the program using the same data used to desk-check the algorithm.

    d. Enter your C++ instructions into a source file named Introductory11.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

    e. Save and run the program. Test the program using the same data used to desk-check the program.

12. A concert hall has three seating categories: Orchestra, Main floor, and Balcony. Orchestra seats are $25. Main floor seats are $30, and Balcony seats are $15. The manager wants a program that allows him to enter the number of tickets sold in each seating category. The program should calculate and display the amount of revenue generated by each seating category, as well as the total revenue. INTRODUCTORY

    a. Create an IPO chart for the problem, and then desk-check the algorithm twice. For the first desk-check, use 50, 100, and 75 as the number of Orchestra, Main floor, and Balcony seats. For the second desk-check, use 30, 25, and 99.

    b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown in Figure 4-12 in the chapter. Then code the algorithm into a program.

    c. Desk-check the program using the same data used to desk-check the algorithm.

    d. Enter your C++ instructions into a source file named Introductory12.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

    e. Save and run the program. Test the program using the same data used to desk-check the program.

INTERMEDIATE

13. The manager of Mama Calari's Pizza Palace wants a program that calculates and displays the number of slices of pizza into which a circular pizza can be divided. The manager will enter the radius of the entire pizza. For this exercise, use the number 14.13 as the area of a pizza slice, and use 3.14 as the value of pi.

    a. Create an IPO chart for the problem, and then desk-check the algorithm twice. For the first desk-check, use 10 as the radius of the pizza. For the second desk-check, use 6. (Hint: For the first desk-check, the number of slices should be a little over 22.)

    b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown in Figure 4-12 in the chapter. Then code the algorithm into a program.

    c. Desk-check the program using the same data used to desk-check the algorithm.

    d. Enter your C++ instructions into a source file named Intermediate13.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

    e. Save and run the program. Test the program using the same data used to desk-check the program.

ADVANCED

14. Complete INTRODUCTORY Exercise 12. Enter (or copy) the instructions from the Introductory12.cpp file into a new source file named Advanced14.cpp. Modify the code in the Advanced14.cpp file so that it also calculates and displays the percentage of the total revenue contributed by each seating category. Save and run the program. Test the program using 50, 100, and 75 as the number of Orchestra, Main floor, and Balcony seats. Then test it using 30, 25, and 99. (Don't be concerned about the extra decimal places in the answers.)

ADVANCED

15. In this exercise, you explore the use of integers in monetary calculations.

    a. Follow the instructions for starting C++ and opening the Advanced15.cpp file. If necessary, make the `system("pause");` statement a comment, and then save the program. Run the program. When you are prompted to enter the gross pay, type 45.13 and press Enter. The net pay that appears on the computer screen is incorrect because it is not the result of subtracting the taxes from the gross pay. Press any key to stop the program.

    b. Review the code contained in the Advanced15.cpp file. The `#include <iomanip>` directive tells the C++ compiler to include the contents of the iomanip file in the current program. The iomanip file contains the definition of the `setprecision` stream manipulator, which appears in the `cout << fixed << setprecision(2) << endl;` statement. The `fixed` stream manipulator is defined in the iostream file, and it forces a real number to display a specific number of decimal places, as specified by the `setprecision` stream manipulator. In this program, the output values will display with two decimal places. You will learn about the directive and both stream manipulators in Chapter 5.

c. Why does the net pay appear as $33.85 rather than $33.84? Hint: Change the `cout << fixed << setprecision(2) << endl;` statement to a comment, and then save and run the program. Type 45.13 as the gross pay and press Enter. Study the output, and then stop the program and change the comment back to a statement.

d. Use the comments that appear in the Advanced15.cpp file to modify the program's code. Why do you need to add .5 to the expressions that calculate the federal and state taxes?

e. Save, run, and test the program to verify that it is working correctly, and then stop the program.

16. Follow the instructions for starting C++ and opening the SwatTheBugs16.cpp file. The program declares and initializes a `double` variable. It then adds 1.5 to the variable before displaying the variable's value. If necessary, make the `system("pause");` statement a comment, and then save the program. Run and then debug the program.

SWAT THE BUGS

Answers to TRY THIS Exercises



Pencil and Paper

1. See Figure 4-38.

IPO chart information	C++ instructions
Input	
sale1	`double sale1 = 0.0;`
sale2	`double sale2 = 0.0;`
commission rate	`double commissionRate = 0.0;`
Processing	
total sales	`double totalSales = 0.0;`
Output	
commission	`double commission = 0.0;`
Algorithm	

1. *enter sale1, sale2, and the commission rate*	`cout << "First sale? ";`
	`cin >> sale1;`
	`cout << "Second sale? ";`
	`cin >> sale2;`
	`cout << "Commission rate in decimal format? ";`
	`cin >> commissionRate;`

2. *calculate the total sales by adding sale1 to sale2*	`totalSales = sale1 + sale2;`
3. *calculate the commission by multiplying the total sales by the commission rate*	`commission = totalSales * commissionRate;`
4. *display the commission*	`cout << "Commission: $" << commission << endl;`

Figure 4-38

2. See Figure 4-39.

IPO chart information	C++ instructions
Input	
cup price	`double cupPrice = 0.0;`
plate price	`double platePrice = 0.0;`
cups purchased	`int cupsPurchased = 0;`
plates purchased	`int platesPurchased = 0;`
sales tax rate (5.5%)	`const double TAX_RATE = .055;`
Processing	
total cup cost	`double totalCupCost = 0.0;`
total plate cost	`double totalPlateCost = 0.0;`
subtotal	`double subtotal = 0.0;`
Output	
total cost	`double totalCost = 0.0;`
Algorithm	

1. enter cup price, plate price, cups purchased, and plates purchased	`cout << "Cup price: ";`
	`cin >> cupPrice;`
	`cout << "Plate price: ";`
	`cin >> platePrice;`
	`cout << "Cups purchased: ";`
	`cin >> cupsPurchased;`
	`cout << "Plates purchased: ";`
	`cin >> platesPurchased;`

2. calculate the total cup cost by multiplying the cups purchased by the cup price	`totalCupCost = cupsPurchased * cupPrice;`
3. calculate the total plate cost by multiplying the plates purchased by the plate price	`totalPlateCost = platesPurchased * platePrice;`
4. calculate the subtotal by adding the total cup cost to the total plate cost	`subtotal = totalCupCost + totalPlateCost;`
5. calculate the total cost by multiplying the subtotal by the sales tax rate and then adding the result to the subtotal	`totalCost = subtotal * TAX_RATE + subtotal;`
6. display the total cost	`cout << "Total cost: $" << totalCost << endl;`

Figure 4-39

Computer

8. See Figure 4-40.

```
 1 //TryThis8.cpp - calculates and displays the commission
 2 //Created/revised by <your name> on <current date>
 3
 4 #include <iostream>
 5 using namespace std;
 6
 7 int main()
 8 {
 9     //declare variables
10     double sale1          = 0.0;
11     double sale2          = 0.0;
12     double commissionRate = 0.0;
13     double totalSales     = 0.0;
14     double commission     = 0.0;
15
16     //enter input items
17     cout << "First sale? ";
18     cin >> sale1;
19     cout << "Second sale? ";
20     cin >> sale2;
21     cout << "Commission rate in decimal format? ";
22     cin >> commissionRate;
23
24     //calculate total sales and commission
25     totalSales = sale1 + sale2;
26     commission = totalSales * commissionRate;
27
28     //display the commission
29     cout << "Commission: $" << commission << endl;
30
31     system("pause");
32     return 0;
33 }   //end of main function
```

your C++ development tool may not require this statement

Figure 4-40

9. See Figure 4-41.

```
 1 //TryThis9.cpp - calculates and displays the total owed
 2 //Created/revised by <your name> on <current date>
 3
 4 #include <iostream>
 5 using namespace std;
 6
 7 int main()
 8 {
 9     //declare variables and named constant
10     int cupsPurchased     = 0;
11     int platesPurchased   = 0;
12     double cupPrice       = 0.0;
13     double platePrice     = 0.0;
14     double totalCupCost   = 0.0;
15     double totalPlateCost = 0.0;
16     double subtotal       = 0.0;
17     double totalCost      = 0.0;
18     const double TAX_RATE = .055;
19
20     //enter input items
21     cout << "Cup price: ";
22     cin >> cupPrice;
23     cout << "Plate price: ";
24     cin >> platePrice;
25     cout << "Cups purchased: ";
26     cin >> cupsPurchased;
27     cout << "Plates purchased: ";
28     cin >> platesPurchased;
29
30     //calculate total cup cost, total plate
31     //cost, the subtotal, and the total cost
32     totalCupCost = cupsPurchased * cupPrice;
33     totalPlateCost = platesPurchased * platePrice;
34     subtotal = totalCupCost + totalPlateCost;
35     totalCost = subtotal * TAX_RATE + subtotal;
36
37     //display total cost
38     cout << "Total cost: $" << totalCost << endl;
39
40     system("pause");
41     return 0;
42 }   //end of main function
```

your C++ development tool may not require this statement

Figure 4-41

CHAPTER 5

The Selection Structure

After studying Chapter 5, you should be able to:

- Include the selection structure in pseudocode and in a flowchart
- Code a selection structure using the `if` statement
- Include comparison operators in a selection structure's condition
- Include logical operators in a selection structure's condition
- Format numeric output

Making Decisions

As you learned in Chapter 1, all computer programs are written using one or more of three basic control structures: sequence, selection, and repetition. You used the sequence structure in the programs you coded in Chapter 4. Recall that, during runtime, the computer processed the instructions in those programs sequentially—in other words, in the order the instructions appeared in the program. Many times, however, a program will need the computer to make a decision before selecting the next instruction to process. A payroll program, for example, typically has the computer determine whether the number of hours an employee worked is greater than 40. The computer then would select either an instruction that computes regular pay only or an instruction that computes regular pay plus overtime pay. Programs that need the computer to make a decision require the use of the selection structure (also called the decision structure). As you learned in Chapter 1, the **selection structure** indicates that a decision (based on some condition) needs to be made, followed by an appropriate action derived from that decision. But how does a programmer determine whether a problem's solution requires a selection structure? The answer to this question is by studying the problem specification. The first problem specification you will examine in this chapter involves Robin, the mechanical woman from Chapter 1. The problem specification and an illustration of the problem are shown in Figure 5-1 along with the solution from Chapter 1. The solution, which is written in pseudocode, requires only the sequence structure. It does not need a selection structure because no decisions need to be made to get Robin from her initial location in the hallway to her ending location in the bedroom.



As you learned in Chapter 2, pseudocode is a tool that programmers use when planning solutions to problems.

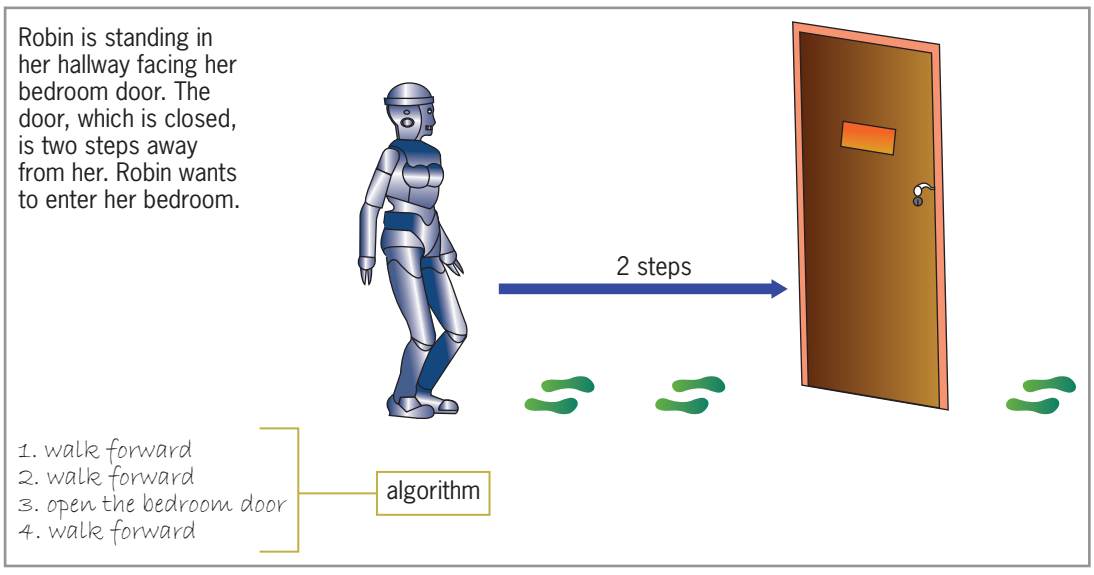

Robin is standing in her hallway facing her bedroom door. The door, which is closed, is two steps away from her. Robin wants to enter her bedroom.
2 steps
1. walk forward
2. walk forward
3. open the bedroom door
4. walk forward
algorithm

Figure 5-1 A problem that requires the sequence structure only

Figure 5-2 shows another problem specification and solution from Chapter 1. The solution in this figure contains both the sequence and selection structures. The selection structure's condition, which is enclosed in parentheses in the pseudocode, directs Robin to make a decision about the status of her bedroom door. More specifically, she needs to determine whether her bedroom door is closed. The condition in a selection structure must be phrased so that it evaluates to a Boolean value: either true or false. In this case, either the bedroom door

is closed (true) or it's not closed (false). If the door is closed, Robin needs to follow the *open the bedroom door* instruction before walking into her bedroom. If the door is not closed, Robin can simply walk into her bedroom. The selection structure in Figure 5-2 is referred to as a **single-alternative selection structure**, because it requires a special action to be taken only when its condition evaluates to true. In this case, the special action is to *open the bedroom door*.

Robin is standing in her hallway facing her bedroom door. The door, which may or may not be closed, is two steps away from her. Robin wants to go inside her bedroom.

```
1. walk forward
2. walk forward
                condition
3. if (the bedroom door is closed)
       open the bedroom door
   end if
4. walk forward
```

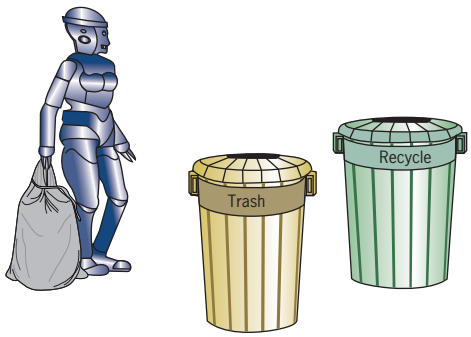
executed only when the condition is true

Figure 5-2 A problem that requires the sequence structure and a single-alternative selection structure

Most programmers use the words *if* and *end if* to denote the beginning and end, respectively, of a selection structure. They also indent the instructions within the selection structure.

Figure 5-3 shows another problem specification and illustration involving Robin, along with the correct solution. As the figure indicates, the solution does not require Robin to make any decisions in order to accomplish her tasks. She needs simply to lift the Trash container's lid, drop the bag of trash in the container, and then put the lid back on the container.

Robin is holding a bag of trash in her right hand. She is directly in front of two containers: one marked Trash and the other marked Recycle. A lid is on each container. Robin needs to lift the Trash container's lid, then drop the bag of trash in the container, and then put the lid back on the container.




```
1. lift the Trash container's lid with your left hand
2. drop the bag of trash in the Trash container
3. put the lid back on the Trash container with your left hand
```

Figure 5-3 Another problem that requires the sequence structure only

Figure 5-4 shows a modified version of the previous problem specification. In the modified version, the contents of the bag that Robin is holding are not certain: the bag might contain trash or it might contain recyclables. You can solve this problem using either of the solutions shown in Figure 5-4. The condition in Solution 1's selection structure determines whether Robin is holding a bag of trash. If she is, she should follow the three instructions contained in the original algorithm from Figure 5-3. If she's not, it means the bag contains recyclables. In that case, Robin should lift the Recycle container's lid before dropping the bag into the container and then replacing the lid. The condition in Solution 2's selection structure, on the other hand, determines whether Robin is holding a bag of recyclables. If she is, the bag belongs in the Recycle container; if she's not, it belongs in the Trash container. Unlike the selection structure in Figure 5-2, which provides a special action for Robin to take only when the selection structure's condition is true, the selection structures in Figure 5-4 require Robin to perform one set of instructions when the condition is true but a different set of instructions when the condition is false. The instructions to follow when the condition evaluates to true are called the **true path**. The true path begins with the *if* and ends with either the *else* (if there is one) or the *end if*. The instructions to follow when the condition evaluates to false are called the **false path**. The false path begins with the *else* and ends with the *end if*. For clarity, the instructions in each path should be indented as shown in Figure 5-4. Selection structures that contain instructions in both paths, like the ones in Figure 5-4, are referred to as **dual-alternative selection structures**.

Robin is holding either a bag of trash or a bag of recyclables in her right hand. She is directly in front of two containers: one marked Trash and the other marked Recycle. A lid is on each container. Robin needs to lift the lid from the appropriate container, then drop the bag in the container, and then put the lid back on the container.

Solution 1

condition

```
if (the bag contains trash)
    lift the Trash container's lid with your left hand
    drop the bag of trash in the Trash container
    put the lid back on the Trash container with your left hand
else
    lift the Recycle container's lid with your left hand
    drop the bag of recyclables in the Recycle container
    put the lid back on the Recycle container with your left hand
end if
```

true path

false path

Solution 2

```
if (the bag contains recyclables)
    lift the Recycle container's lid with your left hand
    drop the bag of recyclables in the Recycle container
    put the lid back on the Recycle container with your left hand
else
    lift the Trash container's lid with your left hand
    drop the bag of trash in the Trash container
    put the lid back on the Trash container with your left hand
end if
```

Figure 5-4 A problem that requires the sequence structure and a dual-alternative selection structure

Flowcharting a Selection Structure

As you learned in Chapter 2, many programmers use flowcharts (rather than pseudocode) when planning solutions to problems. Unlike pseudocode, which consists of short phrases, a flowchart uses standardized symbols to show the steps needed to accomplish a task. Figures 5-5 and 5-6 show two problem specifications along with the correct solutions in flowchart form. The flowchart in Figure 5-5 contains a single-alternative selection structure. You can tell that it's a single-alternative selection structure because it requires a set of actions to be taken only when its condition evaluates to true. Figure 5-6's flowchart contains a dual-alternative selection structure. You can tell that it's a dual-alternative selection structure because it requires two different sets of actions: one to be taken only when its condition evaluates to true, and the other to be taken only when its condition evaluates to false. Recall that the oval in a flowchart is the start/stop symbol, the rectangle is the process symbol, and the parallelogram is the input/output symbol. The diamond in a flowchart is called the **decision symbol**, because it is used to represent the condition (decision) in both the selection and repetition structures. The diamonds in Figures 5-5 and 5-6 represent the condition in a selection structure. (You will learn how to use the diamond to represent a repetition structure's condition in Chapter 7.) The condition in Figure 5-5's diamond checks whether the customer purchased more than five items. It's necessary to do this because the customer receives a 10% discount when more than five items are purchased. The condition in Figure 5-6's diamond, on the other hand, determines whether Mary's sales are at least $15000. In this case, the result (either true or false) determines whether Mary receives a 2% or 1.5% bonus. Notice that the conditions in both diamonds evaluate to either true or false only. Also notice that both diamonds have one flowline entering the symbol and two flowlines leaving the symbol. One of the flowlines leading out of a diamond in a flowchart should be marked with a "T" (for true) and the other should be marked with an "F" (for false). The "T" flowline points to the next instruction to be processed when the condition evaluates to true. In Figure 5-5, the next instruction calculates the 10% discount; in Figure 5-6, it calculates the 2% bonus. The "F" flowline points to the next instruction to be processed when the condition evaluates to false. In Figure 5-5, that instruction displays the total owed; in Figure 5-6, it calculates the 1.5% bonus. You also can mark the flowlines leading out of a diamond with a "Y" and an "N" (for yes and no).

Jerrili's Trading Store wants a program that allows a salesclerk to enter an item's price and the quantity purchased by a customer. The store gives the customer a 10% discount when the quantity purchased is over 5. The program should calculate and display the total amount the customer owes.

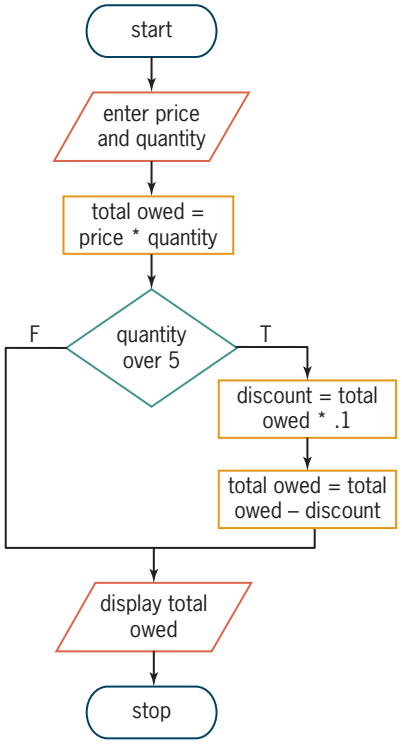


Figure 5-5 Flowchart showing a single-alternative selection structure

Mary Kettleson wants a program that calculates and displays her annual bonus, given her annual sales amount. Mary receives a 2% bonus when her annual sales are at least $15000; otherwise, she receives a 1.5% bonus.

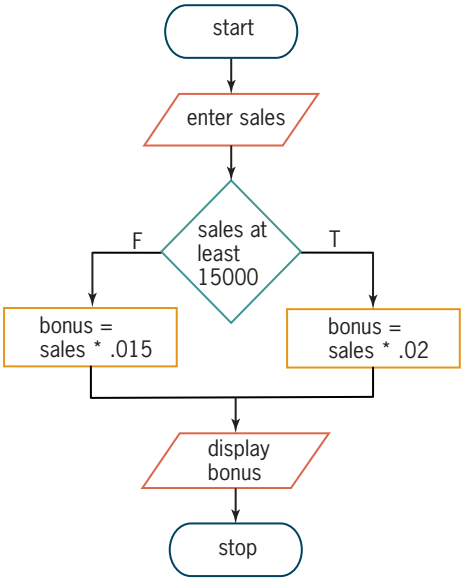


Figure 5-6 Flowchart showing a dual-alternative selection structure

Mini-Quiz 5-1

1. Most programmers use the words _____ to denote the end of a selection structure in pseudocode.

2. The true path in a selection structure can contain only one instruction.
   a. True
   b. False

3. Which of the following is the decision symbol in a flowchart?
   a. diamond
   b. oval
   c. parallelogram
   d. rectangle



The answers to Mini-Quiz questions are located in Appendix A.

Coding a Selection Structure in C++

In most C++ programs, you will use the `if` statement to code a selection structure. The statement's syntax is shown in Figure 5-7. The square brackets in the syntax indicate that the `else` portion, referred to as the `else` clause, is optional. Recall, however, that boldfaced items in a statement's syntax are required. In this case, the keyword `if` and the parentheses that surround the condition are required. The keyword `else` is necessary only in a dual-alternative selection structure. Italicized items in the syntax indicate where the programmer must supply information. In the `if` statement, the programmer must supply the *condition* that the computer needs to evaluate before further processing can occur. The condition must be a Boolean expression, which is an expression that results in a Boolean value (true or false). Besides providing the condition, the programmer must provide the statements to be processed in the true path and (optionally) in the false path. If a path contains more than one statement, the statements must be entered as a **statement block**, which means they must be enclosed in a set of braces (`{}`). Although not a requirement, it is a good programming practice to use a comment (such as `//end if`) to mark the end of the `if` statement in a program. The comment will make your program easier to read and understand. It also will help you keep track of the required `if` and `else` clauses when you nest `if` statements—in other words, when you include one `if` statement inside another `if` statement. You will learn how to nest `if` statements in Chapter 6. The six examples in Figure 5-7 show various ways of using the `if` statement to code selection structures. Examples 1 and 2 are single-alternative selection structures. The remaining four examples are dual-alternative selection structures. Notice that when a path contains multiple statements, the statements are entered as a statement block by enclosing them in braces. Although not shown in Figure 5-7, you also can include the braces even when a path contains only one statement. By doing this, you won't need to remember to enter the braces when statements are added subsequently to the path. Forgetting to enter the braces is a common error made when typing the `if` statement in a program.



C++ also provides the `switch` statement for coding a multiple-alternative selection structure. The `switch` statement is covered in Chapter 6.



In an `if` statement, you cannot have an `else` clause without a matching `if` clause.

HOW TO Use the `if` Statement

Syntax

```
if (condition)
    one or more statements to be processed when the condition is true
[else
    one or more statements to be processed when the condition is false]
//end if
```

Example 1—one statement in only the true path

```
if (condition)
    one statement
//end if
```

Example 2—multiple statements in only the true path

```
if (condition)
{
    multiple statements enclosed in braces
}  //end if
```

Example 3—one statement in each path

```
if (condition)
    one statement
else
    one statement
//end if
```

Example 4—multiple statements in the true path and one statement in the false path

```
if (condition)
{
    multiple statements enclosed in braces
}
else
    one statement
//end if
```

Example 5—one statement in the true path and multiple statements in the false path

```
if (condition)
    one statement
else
{
    multiple statements enclosed in braces
}  //end if
```

Example 6—multiple statements in both paths

```
if (condition)
{
    multiple statements enclosed in braces
}
else
{
    multiple statements enclosed in braces
}  //end if
```

Figure 5-7 How to use the `if` statement

As mentioned earlier, an `if` statement's condition must be a Boolean expression, which is an expression that evaluates to either true or false. The expression can contain variables, constants, arithmetic operators, comparison operators, and logical operators. You already know about variables, constants, and arithmetic operators. You will learn about comparison operators and logical operators in this chapter.

Comparison Operators

The C++ operators listed in Figure 5-8 are called **comparison operators**, because they are used to compare two values that have the same data type. The precedence numbers in the figure indicate the order in which the computer performs comparisons in an expression. Comparisons with a precedence number of 1 are performed before comparisons with a precedence number of 2; however, you can use parentheses to override the order of precedence. Expressions containing a comparison operator always evaluate to a Boolean value: either true or false. Notice that four of the C++ comparison operators contain two symbols. When entering these operators, be sure you do not enter a space between the symbols and be sure to enter both symbols in the exact order shown in Figure 5-8. Also included in Figure 5-8 are examples of using comparison operators in an `if` statement's condition.



Comparison operators also are referred to as relational operators.

HOW TO Use Comparison Operators in an `if` Statement's Condition

Operator	Operation	Precedence number
<	less than	1
<=	less than or equal to	1
>	greater than	1
>=	greater than or equal to	1
==	equal to	2
!=	not equal to	2

Examples (All of the variables have the `int` data type.)

```
if (quantity < 50)
if (age >= 25)
if (onhand == target)
if (quantity != 7500)
```



Keep in mind that == is the opposite of !=, > is the opposite of <=, and < is the opposite of >=.

Figure 5-8 How to use comparison operators in an `if` statement's condition

As Figure 5-8 indicates, you use two equal signs (==) to test for equality in C++. To test for inequality, you use an exclamation point (which stands for *not*) followed by an equal sign, like this: `!=`. Keep in mind that you should never use either the equality operator (==) or the inequality operator (!=) to compare two real numbers. Because some real numbers cannot be stored precisely in memory, the numbers should never be compared for equality or inequality. Instead, you should test that the difference between the real numbers you are comparing is less than some acceptable small value, such as .00001. You will learn how to determine whether two real numbers are equal in Computer Exercise 15 at the end of this chapter.



Numbers are compared using their binary equivalents.

When an expression contains more than one comparison operator with the same precedence number, the computer evaluates the comparison operators from left to right in the expression, similar to what is done with arithmetic operators. Comparison operators are evaluated after any arithmetic operators in an expression. For example, when processing the expression `5 - 2 > 1 + 2`, the computer will evaluate the two arithmetic operators before it evaluates the comparison operator. The result of the expression is the Boolean value false, as shown in Figure 5-9.

Original expression	5 – 2 > 1 + 2
The subtraction is performed first	3 > 1 + 2
The addition is performed next	3 > 3
The > comparison is performed last	false

Figure 5-9 Evaluation steps for an expression containing arithmetic and comparison operators

It is easy to confuse the equality operator (==) with the assignment operator (=). You use the equality operator to compare two values to determine whether they are equal, as in the condition in the following `if` clause: `if (num == 1)`. You use the assignment operator, on the other hand, to assign a value to a memory location. An example of this is the statement `num = 1;`. In the next two sections, you will view programs that contain a comparison operator in an `if` statement's condition.

Swapping Numeric Values

Figure 5-10 shows the IPO chart information and C++ code for a program that displays the lowest and highest of two integers entered by the user. (The flowchart for this program is contained in the Ch5Flowcharts.pdf file, which is located in the Cpp6\Chap05 folder on your computer's hard drive or on the device designated by your instructor.) The program contains a single-alternative selection structure. The `firstNum > secondNum` condition in the `if` clause compares the contents of the `firstNum` variable with the contents of the `secondNum` variable. If the value in the `firstNum` variable is greater than the value in the `secondNum` variable, the condition evaluates to true and the four instructions in the `if` statement's true path swap both values. Swapping the values places the smaller number in the `firstNum` variable and places the larger number in the `secondNum` variable. Notice that the four instructions in the true path are enclosed in braces. As you learned earlier, when more than one instruction needs to be processed when the `if` statement's condition is true, the C++ syntax requires those instructions to be entered as a statement block. If the condition in the `if` clause in the program evaluates to false, the instructions in the true path are skipped over because the `firstNum` variable already contains a number that is smaller than (or possibly equal to) the number stored in the `secondNum` variable.

IPO chart information	C++ instructions
Input	
first number	`int firstNum = 0;`
second number	`int secondNum = 0;`
Processing	
none	
Output	
first number (lowest)	
second number (highest)	
Algorithm	

1. enter the first number and second number	`cout << "Enter an integer: ";`
	`cin >> firstNum;`
	`cout << "Enter another integer: ";`
	`cin >> secondNum;`

2. if (the first number is greater than the second number)
swap the numbers so that the first number is the lowest number
//end if
	`if (firstNum > secondNum)`
	`{`
	`int temp = 0;`
	`temp = firstNum;`
	`firstNum = secondNum;`
	`secondNum = temp;`
	`}  //end if`

3. display the first number and the second number	`cout << "Lowest: " <<`
	`firstNum << endl;`
	`cout << "Highest: " <<`
	`secondNum << endl;`

Figure 5-10 IPO chart information and C++ instructions for the swapping program

Study closely the instructions used to swap the values stored in the `firstNum` and `secondNum` variables. The instructions are shaded in Figure 5-10. The first instruction, `int temp = 0;`, declares and initializes a variable named `temp`. Because the `temp` variable is declared in the `if` statement's true path, it can be used only by the instructions within that path. More specifically, it can be used only by the instructions that follow the declaration statement within the true path. A variable that can be used only within the statement block in which it is defined is referred to as a **local variable**. In this case, the `temp` variable is local to the `if` statement's true path. You may be wondering why the `temp` variable was not declared at the beginning of the `main` function, along with the `firstNum` and `secondNum` variables. Although there is nothing wrong with declaring the `temp` variable in that location, there is no reason to create the variable until it is needed, which is only when a swap is necessary. The second instruction in the `if` statement's true path, `temp = firstNum;`, assigns the value in the `firstNum` variable to the `temp` variable. If you do not store the `firstNum` variable's value in the `temp` variable, the value will be lost when the computer processes the next statement, `firstNum = secondNum;`, which replaces the contents of the `firstNum` variable with the contents of the `secondNum` variable. Finally, the `secondNum = temp;` instruction assigns the `temp` variable's value to the `secondNum` variable; this completes the swap. Figure 5-11 illustrates the concept of swapping, assuming the user enters the numbers 10 and 3. Figure 5-12 shows a sample run of the number swapping program.

	firstNum	secondNum	temp
values stored in the variables after the `cin` and `int temp = 0;` statements are processed	10	3	0
result of the `temp = firstNum;` statement	10	3	10
result of the `firstNum = secondNum;` statement	3	3	10
result of the `secondNum = temp;` statement	3	10	10

the values were swapped

Figure 5-11 Illustration of the swapping concept

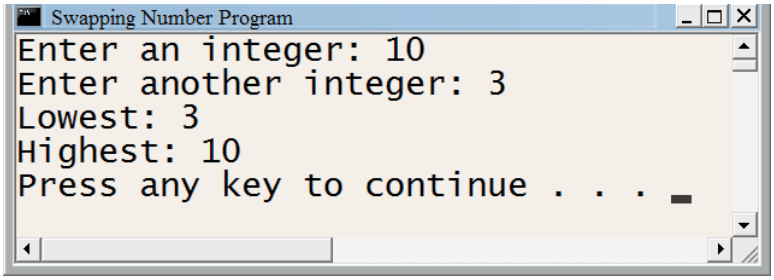


Figure 5-12 Sample run of the number swapping program

Displaying the Sum or Difference

Figure 5-13 shows the IPO chart information and C++ code for a program that displays either the sum of or difference between two numbers entered by the user. The program contains a dual-alternative selection structure. (The flowchart for this program is contained in the Ch5Flowcharts.pdf file, which is located in the Cpp6\Chap05 folder on your computer's hard drive or on the device designated by your instructor.) The program uses an `int` variable named `operation` and three `double` variables named `janSales`, `febSales`, and `answer`. The program prompts the user to enter either the number 1 (for addition) or the number 2 (for subtraction); it stores the user's response in the `operation` variable. The program then prompts the user to enter two sales amounts, which it stores in the `janSales` and `febSales` variables. The `operation == 1` condition in the `if` clause compares the contents of the `operation` variable with the number 1. The condition will evaluate to true only when the user enters the number 1. It will evaluate to false when the user enters anything other than the number 1. If the condition evaluates to true, the instructions in the selection structure's true path calculate and display the sum of the sales amounts stored in the `janSales` and `febSales` variables. If the condition evaluates to false, the instructions in the selection structure's false path calculate and display the difference between the two sales amounts. Notice that the instructions in each path are entered as a statement block. Figure 5-14 shows a sample run of the sum or difference program.

IPO chart information	C++ instructions
Input	
operation	`int operation = 0;`
January sales	`double janSales = 0.0;`
February sales	`double febSales = 0.0;`
Processing	
none	
Output	
either the sum of or the difference between the January sales and February sales	`double answer = 0.0;`
Algorithm	

1. enter the operation, January sales, and February sales	`cout << "Enter 1 (add) or 2 (subtract): ";`
	`cin >> operation;`
	`cout << "January sales: ";`
	`cin >> janSales;`
	`cout << "February sales: ";`
	`cin >> febSales;`

2. if (the operation is 1)	`if (operation == 1)` `{`
calculate the answer by adding the January sales to the February sales	`answer = janSales + febSales;`
display "Sum:" and the answer	`cout << "Sum: " <<` `answer << endl;`
	`}`
else	`else` `{`
calculate the answer by subtracting the February sales from the January sales	`answer = janSales - febSales;`
display "Difference:" and the answer	`cout << "Difference: " <<` `answer << endl;`
end if	`} //end if`

Figure 5-13 IPO chart information and C++ instructions for the sum or difference program

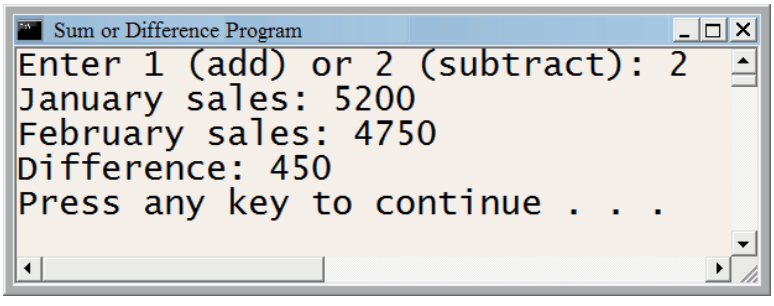

Sum or Difference Program
Enter 1 (add) or 2 (subtract): 2
January sales: 5200
February sales: 4750
Difference: 450
Press any key to continue . . .

Figure 5-14 Sample run of the sum or difference program



The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 5-2

1. You create a statement block by enclosing one or more statements in _____.
   a. parentheses
   b. square brackets
   c. quotation marks
   d. none of the above

2. Which of the following determines whether an `int` variable named `quantity` contains the number 100?
   a. `if (quantity = 100)`
   b. `if quantity = 100`
   c. `if (quantity == 100)`
   d. none of the above

3. Which of the following determines whether the value contained in the `sales` variable is at least $300.99?
   a. `if (sales >= 300.99)`
   b. `if (sales => 300.99)`
   c. `if (sales > = 300.99)`
   d. both a and c

4. Which of the following is the inequality operator in C++?
   a. &=
   b. ≠
   c. `!=`
   d. none of the above

Logical Operators

You also can use logical operators in an `if` statement's condition. **Logical operators** allow you to combine two or more conditions, referred to as sub-conditions, into one compound condition. Logical operators sometimes are referred to as **Boolean operators**, because the compound condition in which they are contained always evaluates to either true or false only. Figure 5-15 lists the two most commonly used logical operators: And and Or. When the And logical operator is used to create a compound condition, all of the sub-conditions must be true for the compound condition to be true. However, when the Or logical operator is used, only one of the sub-conditions must be true for the compound condition to be true. C++ uses special symbols to represent the And and Or logical operators in a program. The And operator in C++ is two ampersands (&&); the Or operator is two pipe symbols (||).

On most computer keyboards, the pipe symbol (|) is located on the same key as the backslash (\). Figure 5-15 also shows the order of precedence for the And and Or operators, as well as examples of using the operators in an `if` statement's condition. Notice that the compound condition in each example evaluates to either true or false only. Logical operators are evaluated after any arithmetic or comparison operators in an expression.

HOW TO Use Logical Operators in an `if` Statement's Condition

Operator	Operation	Precedence number
And (&&)	all sub-conditions must be true for the compound condition to evaluate to true	1
Or (||)	only one of the sub-conditions needs to be true for the compound condition to evaluate to true	2

Example 1

```
int quantity = 0;
cin >> quantity;
if (quantity > 0 && quantity < 50)
```

The compound condition evaluates to true when the number stored in the `quantity` variable is greater than zero and, at the same time, less than 50; otherwise, it evaluates to false.

Example 2

```
int age = 0;
cin >> age;
if (age == 21 || age > 55)
```

The compound condition evaluates to true when the number stored in the `age` variable is either equal to 21 or greater than 55; otherwise, it evaluates to false.

Example 3

```
int quantity = 0;
double price = 0.0;
cin >> quantity;
cin >> price;
if (quantity < 100 && price < 10.35)
```

The compound condition evaluates to true when the number stored in the `quantity` variable is less than 100 and, at the same time, the number stored in the `price` variable is less than 10.35; otherwise, it evaluates to false.

Example 4

```
int quantity = 0;
double price = 0.0;
cin >> quantity;
cin >> price;
if (quantity > 0 && quantity < 100 || price > 34.55)
```

The compound condition evaluates to true when either (or both) of the following is true: the number stored in the `quantity` variable is between 0 and 100 or the number stored in the `price` variable is greater than 34.55; otherwise, it evaluates to false. (The && operator is evaluated before the || operator, because it has a higher precedence.)

Figure 5-15 How to use logical operators in an `if` statement's condition

As mentioned earlier, all expressions containing a logical operator result in an answer of either true or false only. The tables shown in Figure 5-16, called **truth tables**, summarize how the computer evaluates the logical operators in an expression. As the figure indicates, when you use the And operator to combine two sub-conditions (sub-condition1 && sub-condition2), the resulting compound condition is true only when both sub-conditions are true. If either condition is false or if both conditions are false, then the compound condition is false. When you use the Or operator to combine two sub-conditions (sub-condition1 || sub-condition2), the compound condition is false only when both sub-conditions are false. If either sub-condition is true or if both sub-conditions are true, then the compound condition is true.

Truth table for the And (&&) operator

sub-condition1	sub-condition2	sub-condition1 && sub-condition2
true	true	true
true	false	false
false	true (not evaluated)	false
false	false (not evaluated)	false

Truth table for the Or (||) operator

sub-condition1	sub-condition2	sub-condition1 || sub-condition2
true	true (not evaluated)	true
true	false (not evaluated)	true
false	true	true
false	false	false

Figure 5-16 Truth tables for the logical operators

As indicated in Figure 5-16, when the computer evaluates the "sub-condition1 && sub-condition2" expression, it will not evaluate sub-condition2 when sub-condition1 is false. Because both sub-conditions combined with the And operator need to be true for the compound condition to be true, there is no need to evaluate sub-condition2 when sub-condition1 is false. When the computer evaluates the "sub-condition1 || sub-condition2" expression, on the other hand, it will not evaluate sub-condition2 when sub-condition1 is true. Because only one of the sub-conditions combined with the Or operator needs to be true for the compound condition to be true, there is no need to evaluate sub-condition2 when sub-condition1 is true. The concept of evaluating sub-condition2 based on the result of sub-condition1 is referred to as **short-circuit evaluation**.

Using the Truth Tables



Notice that the values being compared in each sub-condition have the same data type.

A program needs to calculate a bonus for each A-rated salesperson whose monthly sales total more than $5000. The program uses a `char` variable named `rating` and an `int` variable named `sales` to store the salesperson's rating and sales amount, respectively. Therefore, you can phrase sub-condition1 as `rating == 'A'` and phrase sub-condition2 as `sales > 5000`.

Which logical operator should you use to combine both sub-conditions into one compound condition? You can use the truth tables from Figure 5-16 to help you answer this question. For a salesperson to receive a bonus, both sub-condition1 and sub-condition2 must be true at the same time. If either condition is false or if both conditions are false, then the compound condition should be false and the salesperson should not receive a bonus. According to the truth tables, both logical operators evaluate a compound condition as true when both sub-conditions are true. However, only the And operator evaluates a compound condition as false when either one or both of the sub-conditions are false. Therefore, the correct compound condition to use in this case is `rating == 'A' && sales > 5000`.

Now assume you want to send a letter to all A-rated salespeople and all B-rated salespeople. If the rating is stored in the `rating` variable, you can phrase sub-condition1 as `rating == 'A'` and phrase sub-condition2 as `rating == 'B'`. Now which logical operator should you use to combine both sub-conditions? At first it might appear that the And operator is the correct one to use, because the example says to send the letter to "all A-rated salespeople and all B-rated salespeople." In everyday conversations, people sometimes use the word *and* when what they really mean is *or*. Although both words do not mean the same thing, using *and* instead of *or* generally does not cause a problem, because we are able to infer what another person means. Computers, however, cannot infer anything; they simply process the directions you give them, word for word. In this case, you actually want to send a letter to all salespeople with either an A or a B rating (a salesperson can have only one rating), so you need to use the Or operator. As the truth tables indicate, the Or operator is the only operator that evaluates the compound condition as true when at least one of the sub-conditions is true. Therefore, the correct compound condition to use in this case is `rating == 'A' || rating == 'B'`. In the next two sections, you will view programs that contain a logical operator in an `if` statement's condition.

Calculating Gross Pay

A program needs to calculate and display an employee's gross pay. To keep this example simple, no one at the company works more than 40 hours per week and everyone earns the same hourly rate, $10. Before making the gross pay calculation, the program should verify that the number of hours entered by the user is greater than or equal to 0 but less than or equal to 40. Programmers refer to the process of verifying that the input data is within the expected range as **data validation**. If the number of hours is valid, the program should calculate and display the gross pay. Otherwise, it should display an error message alerting the user that the number of hours is incorrect. You can use either of the examples in Figure 5-17 to code the program. Both examples contain a dual-alternative selection structure. Notice that the compound condition in Example 1 uses the And operator, whereas the compound condition in Example 2 uses the Or operator. Example 1's compound condition determines whether the value stored in the `hoursWorked` variable is greater than or equal to 0 and (at the same time) less than or equal to 40. If the compound condition evaluates to true, the selection structure both calculates and displays the gross pay; otherwise, it displays the "Incorrect number of hours" message. Example 2's compound condition determines whether

the value stored in the `hoursWorked` variable is either less than 0 or greater than 40. If the compound condition evaluates to true, the selection structure displays the "Incorrect number of hours" message; otherwise, it both calculates and displays the gross pay. Both examples in Figure 5-17 produce the same results and simply represent two different ways of performing the same task. Figures 5-18 and 5-19 show sample runs of a program that contains the code shown in either of the examples in Figure 5-17.

Example 1

```
//declare constant and variables
const int PAY_RATE = 10;
int hoursWorked    = 0;
int grossPay       = 0;

//enter input items
cout << "Hours worked (0 through 40): ";
cin >> hoursWorked;

//calculate and display output
if (hoursWorked >= 0 && hoursWorked <= 40)
{
    grossPay = hoursWorked * PAY_RATE;
    cout << "Gross pay: $" << grossPay << endl;
}
else
    cout << "Incorrect number of hours" << endl;
//end if
```

And operator

Example 2

```
//declare constant and variables
const int PAY_RATE =  10;
int hoursWorked    =  0;
int grossPay       =  0;

//enter input items
cout << "Hours worked (0 through 40): ";
cin >> hoursWorked;

//calculate and display output

if (hoursWorked < 0 || hoursWorked > 40)
    cout << "Incorrect number of hours" << endl;
else
{
    grossPay = hoursWorked * PAY_RATE;
    cout << "Gross pay: $" << grossPay << endl;
}   //end if
```

Or operator

Figure 5-17 Examples of C++ instructions for the gross pay program

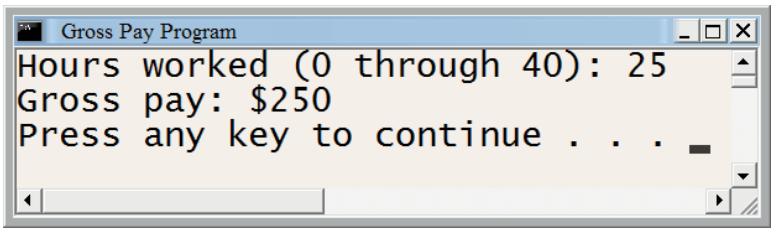


Figure 5-18 First sample run of the gross pay program's code

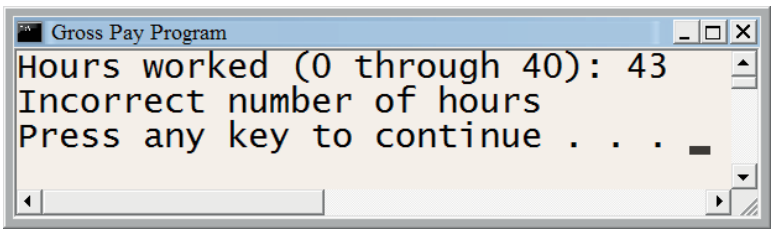


Figure 5-19 Second sample run of the gross pay program's code

Pass/Fail Program

A program needs to display the word "Pass" when the user enters the letter P (in either uppercase or lowercase) and display the word "Fail" when the user enters anything else. You can use either of the examples in Figure 5-20 to code the program. Both examples contain a dual-alternative selection structure. The compound condition in Example 1 uses the Or operator to determine whether the `letter` variable contains either the uppercase letter P or the lowercase letter p. When the variable contains one of those two letters, the compound condition evaluates to true and the selection structure displays the word "Pass" on the screen; otherwise, it displays the word "Fail". You may be wondering why you need to compare the contents of the `letter` variable with both the uppercase and lowercase forms of the letter P. As is true in many programming languages, character comparisons in C++ are case sensitive, which means that the uppercase version of a letter is not the same as its lowercase counterpart. So, although a human being recognizes P and p as being the same letter, a computer does not; to a computer, a P is different from a p. You learned the reason for this differentiation in Chapter 3. Recall that each character on a computer keyboard is assigned a unique ASCII code, which is stored in the computer's internal memory using a group of 0s and 1s. The ASCII code for the uppercase letter P is 80 and is stored using the eight bits 01010000. The ASCII code for the lowercase letter p, on the other hand, is 112 and is stored using the eight bits 01110000. Instead of using the Or operator to determine whether the `letter` variable contains either P or p, you also can use the And operator, as shown in Example 2 in Figure 5-20. The compound condition in the second example determines whether the value in the `letter` variable is not equal to the uppercase letter P and (at the same time) not equal to the lowercase letter p. When the variable does not contain either of those two letters, the compound condition evaluates to true and the selection structure displays the word "Fail" on the screen; otherwise, it displays the word "Pass". Figures 5-21 and 5-22 show sample runs of a program that contains the code shown in either of the examples in Figure 5-20.



As you learned in Chapter 3, character literal constants are enclosed in single quotation marks, like this: 'P'. String literal constants are enclosed in double quotation marks, like this: "Pass".



The full ASCII chart is contained in Appendix C in this book.

Or operator

And operator

```
Example 1
//declare variable
char letter = ' ';

//enter input item, then display message
cout << "Enter a letter: ";
cin >> letter;

if (letter == 'P' || letter == 'p')
    cout << "Pass" << endl;
else
    cout << "Fail" << endl;
//end if

Example 2
//declare variable
char letter = ' ';

//enter input item, then display message
cout << "Enter a letter: ";
cin >> letter;

if (letter != 'P' && letter != 'p')
    cout << "Fail" << endl;
else
    cout << "Pass" << endl;
//end if
```

Figure 5-20 Examples of C++ instructions for the Pass/Fail program

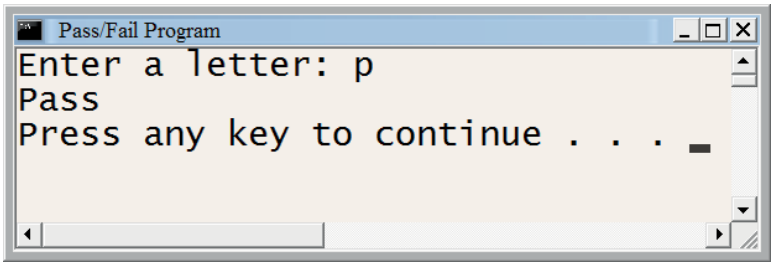


Figure 5-21 First sample run of the Pass/Fail program's code

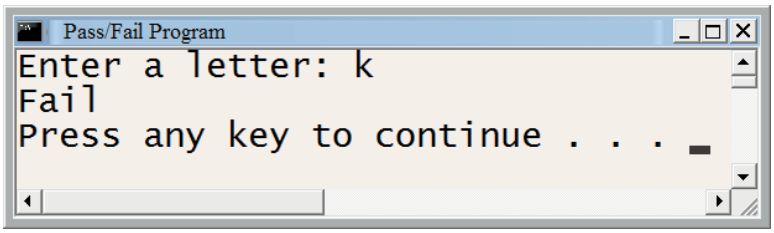


Figure 5-22 Second sample run of the Pass/Fail program's code

Figure 5-23 shows the order of precedence for the arithmetic, comparison, and logical operators you have learned so far. Recall that operators with the same precedence number are evaluated from left to right in an expression. Notice that logical operators are evaluated after any arithmetic operators or comparison operators in an expression. As a result, when the computer processes the expression `30 > 75 / 3 && 5 < 10 * 2`, it evaluates the arithmetic operators first, followed by the comparison operators, followed by the logical operator. The expression evaluates to true, as shown in the example included in Figure 5-23.

Operator	Operation	Precedence number
( )	override normal precedence rules	1
–	negation (reverses the sign of a number)	2
*, /, %	multiplication, division, and modulus arithmetic	3
+, –	addition and subtraction	4
<, <=, >, >=	less than, less than or equal to, greater than, greater than or equal to	5
==, !=	equal to, not equal to	6
And (&&)	all sub-conditions must be true for the compound condition to evaluate to true	7
Or (||)	only one of the sub-conditions needs to be true for the compound condition to evaluate to true	8

Example

Original expression	30 > 75 / 3 && 5 < 10 * 2
75 / 3 is performed first	30 > 25 && 5 < 10 * 2
10 * 2 is evaluated second	30 > 25 && 5 < 20
30 > 25 is evaluated third	true && 5 < 20
5 < 20 is evaluated fourth	true && true
true && true is evaluated last	true

Figure 5-23 Listing and an example of arithmetic, comparison, and logical operators

Mini-Quiz 5-3



The answers to Mini-Quiz questions are located in Appendix A.

1. The compound condition `true || false` will evaluate to _____.

2. The compound condition `7 > 3 && 5 < 2` will evaluate to _____.

3. The compound condition `5 * 4 < 20 || false` will evaluate to _____.

4. Which of the following `if` clauses determines whether the value in an `int` variable named `age` is between 30 and 40, including 30 and 40?
   a. `if (age <= 30 || age >= 40)`
   b. `if (age >= 30 && age <= 40)`
   c. `if (age >=30 && <= 40)`
   d. `if (age <=30 || >= 40)`

5. Which of the following `if` clauses determines whether a `char` variable named `code` contains the letter R (in any case)?
   a. `if (code == 'R' || code == 'r')`
   b. `if (code = 'R' || code = 'r')`
   c. `if (code == "R" || code == "r")`
   d. none of the above

Converting a Character to Uppercase or Lowercase

Earlier, in Figure 5-20, you viewed two examples of C++ code for the Pass/Fail program. In both examples, the `if` clause contains a compound condition that compares the character stored in the `letter` variable with both the uppercase and lowercase forms of the letter P. However, that is not the only way of comparing a variable's contents to both versions of a letter; you also can use either of the following C++ built-in functions: `toupper` or `tolower`. The **toupper function** temporarily converts a character to uppercase, while the **tolower function** converts it to lowercase. Figure 5-24 shows each function's syntax. An item that appears between parentheses in a function's syntax is called an **argument**, and it represents information that the function needs to perform its task. In this case, the `toupper` and `tolower` functions need the name of a variable whose data type is `char`. Both functions copy the character stored in the *charVariable* to a temporary location in the computer's internal memory. The functions convert the temporary character to the appropriate case (if necessary) and then return the temporary character. Keep in mind that the `toupper` and `tolower` functions do not change the contents of the *charVariable*; they change the contents of the temporary location only. In addition, the `toupper` and `tolower` functions affect only characters that represent letters of the alphabet, as these are the only characters that have uppercase and lowercase forms. Also included in Figure 5-24 are examples of using the `toupper` and `tolower` functions. When using the `toupper` function in a comparison, be sure that everything you are comparing is uppercase; otherwise, the comparison will not evaluate correctly. For instance, the clause `if (toupper(letter) == 'p')` is not correct: the condition will always evaluate to false, because the uppercase version of a letter will never be equal to its lowercase counterpart. Likewise, when using the `tolower` function in a comparison, be sure that everything you are comparing is lowercase.



Some programmers pronounce `char` as "care" because it is short for *character*, while others pronounce `char` as in the first syllable of the word *charcoal*.

HOW TO Use the `toupper` and `tolower` Functions

Syntax
toupper(*charVariable***)**
tolower(*charVariable***)**

Example 1
```
if (toupper(letter) == 'P')
```
The condition compares the uppercase character returned by the `toupper` function with the uppercase letter P. The condition evaluates to true when the character stored in the `letter` variable is either P or p.

Example 2
```
if (tolower(letter) == 'p')
```
The condition compares the lowercase character returned by the `tolower` function with the lowercase letter p. The condition evaluates to true when the character stored in the `letter` variable is either P or p.

Example 3
```
initial = toupper(initial);
```
The assignment statement changes the contents of the `initial` variable to uppercase.

Figure 5-24 How to use the `toupper` and `tolower` functions

Formatting Numeric Output

In a C++ program, numbers with a decimal place are displayed in either fixed-point or e (exponential) notation, depending on the size of the number. Recall that a number with a decimal place is called a real number. Smaller real numbers—those containing six or fewer digits to the left of the decimal point—usually are displayed in fixed-point notation. For example, the number 1,234.56 would be displayed in fixed-point notation as 1234.560000. Larger real numbers—those containing more than six digits to the left of the decimal point—typically are displayed in e notation. The number 1,225,000.00, for example, would be displayed in e notation as 1.225e+006. The type of program you are creating determines the appropriate format to use when displaying numbers with a decimal place. Business programs usually display real numbers in fixed-point notation, while many scientific programs use e notation. C++ provides stream manipulators that allow you to control the format used to display real numbers. You use the **fixed stream manipulator** to display real numbers in fixed-point notation. To display real numbers in e notation, you use the **scientific stream manipulator**. The appropriate manipulator must appear in a `cout` statement, and it must be processed before the real numbers you want formatted are displayed. After being processed, the manipulator remains in effect either until the end of the program or until the computer encounters another manipulator that changes the format, whichever occurs first. Figure 5-25 shows examples of using both manipulators. As the examples indicate, a stream manipulator can appear by itself in a `cout` statement; or, it can be included with other information in a `cout` statement.

HOW TO Use the `fixed` and `scientific` Stream Manipulators

Example 1 — Result

```
double sales = 10575.25;
cout << fixed;
cout << sales << endl;
```

displays 10575.250000

Example 2

```
double rate = 5.12345623;
cout << fixed << rate << endl;
```

displays 5.123456

Example 3

```
double rate = 5.123456932;
cout << fixed << rate << endl;
```

displays 5.123457

Example 4

```
double sales = 10575.25;
cout << scientific << sales << endl;
```

displays 1.057525e+004

Figure 5-25 How to use the `fixed` and `scientific` stream manipulators

Study closely the examples in Figure 5-25. Notice that the code in Example 1 displays 10575.250000 rather than 10575.25. This is because all real numbers formatted by the `fixed` stream manipulator will have six digits to the right of the decimal point. If the unformatted number contains less than six decimal places, the `fixed` stream manipulator pads the number with zeros until it has six decimal places. The number 10575.25, for instance, is padded with four zeros to make 10575.250000. If the unformatted number contains more than six decimal places, the additional decimal places are truncated (dropped off). Before the truncation occurs, however, the number in the sixth decimal place is either rounded up one number or left as is, depending on the value of the numbers being truncated. For example, when the `rate` variable contains the number 5.12345623, as it does in Example 2, the `cout` statement displays the number 5.123456. Notice that the digits 2 and 3, which occupy the seventh and eighth decimal places in the number 5.12345623, are truncated. Also notice that the number in the sixth decimal place (6) remains the same; no rounding occurs because the value of the remaining numbers (23) is less than 50. However, when the `rate` variable contains the number 5.123456932, as it does in Example 3, the `cout` statement displays 5.123457. In this case, the digits in the seventh, eighth, and ninth decimal places (9, 3, and 2) are truncated. However, because the value of those numbers (932) is greater than 500, the number in the sixth decimal place (6) is rounded up to 7. The `cout << scientific << sales << endl;` statement in Example 4 displays the contents of the `sales` variable in e notation; the result is 1.057525e+004.

In most programs, especially business programs, numeric output is displayed with either zero or two decimal places. Rarely does a program require numbers to be displayed with the six decimal places you get from the `fixed` stream manipulator. You can use the C++ **setprecision stream manipulator** to control the number of decimal places that appear when a real number is displayed. The definition of the `setprecision` manipulator is contained in the iomanip file, which comes with your C++ compiler. (The "io" stands for "input/output".) For a program to use the `setprecision` manipulator,

it must contain the `#include <iomanip>` directive. Figure 5-26 shows the `setprecision` manipulator's syntax. The *numberOfDecimalPlaces* argument in the syntax is an integer that specifies the number of decimal places to include when displaying a real number. The `setprecision` manipulator remains in effect either until the end of the program or until the computer encounters another `setprecision` manipulator. Also included in Figure 5-26 are examples of using the `setprecision` manipulator in a C++ statement. As Example 2 shows, you can include the `setprecision` and `fixed` manipulators in the same statement.



Stream manipulators with arguments (such as `setprecision`) are defined in the iomanip file. Stream manipulators that do not have arguments (such as `fixed` and `scientific`) are defined in the iostream file.

HOW TO Use the `setprecision` Stream Manipulator

Syntax
setprecision(*numberOfDecimalPlaces***)**

Example 1

```
double sales = 3500.6;
cout << fixed;
cout << setprecision(2);
cout << sales << endl;
```

Result: displays 3500.60

Example 2

```
double rate = 10.0732;
cout << fixed << setprecision(3);
cout << rate << endl;
```

Result: displays 10.073

Example 3

```
double sales = 3467.55;
cout << fixed;
cout << setprecision(0) << sales;
```

Result: displays 3468

Figure 5-26 How to use the `setprecision` stream manipulator

Mini-Quiz 5-4



The answers to Mini-Quiz questions are located in Appendix A.

1. Which of the following tells the computer to display real numbers in fixed-point notation with two decimal places?

   a. `cout << fixed << decimal(2);`

   b. `cout << fixedPoint << precision(2);`

   c. `cout << fixedPoint << setdecimal(2);`

   d. `cout << fixed << setprecision(2);`

2. Which of the following changes the contents of a `char` variable named `letter` to lowercase?

   a. `tolower(letter) = letter;`

   b. `letter == tolower(letter);`

   c. `letter = tolower(letter);`

   d. `tolower('letter');`

3. If the `num` variable contains the number 34.65, the `cout << fixed << num;` statement will display the number as _____.

   a. 34.65

   b. 34.650

   c. 34.6500

   d. 34.650000



The answers to the labs are located in Appendix A.



LAB 5-1 Stop and Analyze

Study the program shown in Figure 5-27, and then answer the questions.

```
 1  //Lab5-1.cpp - displays an employee's new salary
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <iomanip>
 6  using namespace std;
 7
 8  int main()
 9  {
10     double salary = 0.0;
11     double rate   = 0.0;
12     char payGrade = ' ';
13
14     cout << "Current salary: ";
15     cin >> salary;
16     cout << "Pay grade (1, 2, or 3): ";
17     cin >> payGrade;
18
19     if (payGrade == '1')
20         rate = .03;
21     else
22         rate = .02;
23     //end if
24
25     salary = salary + salary * rate;
26     cout << fixed << setprecision(2);
27     cout << "New salary: " << salary << endl;
28
29     system("pause");
30     return 0;
31  }  //end of main function
```

Figure 5-27 Program for Lab 5-1

QUESTIONS

1. What rate will be assigned to the `rate` variable when the user enters the following pay grades: 1, 3, and 5?
2. Why is the directive on Line 5 necessary?
3. Why is the 1 in the `if` statement's condition on Line 19 enclosed in single quotation marks?
4. How would you rewrite the `if` statement on Lines 19 through 23 to use the `!=` operator in the condition?
5. How else could you write the `salary = salary + salary * rate;` statement?
6. What changes would you need to make to the program so that it doesn't use the `rate` variable?



LAB 5-2 Plan and Create

In this lab, you will plan and create an algorithm for the manager of Willow Springs Health Club. The problem specification is shown in Figure 5-28.

> The manager of Willow Springs Health Club wants a program that allows her to enter the number of calories and grams of fat contained in a specific food. The program should calculate and display two values: the food's fat calories and its fat percentage. The fat calories are the number of calories attributed to fat. The fat percentage is the ratio of the food's fat calories to its total calories. You can calculate a food's fat calories by multiplying its fat grams by the number 9, because each gram of fat contains 9 calories. To calculate the fat percentage, you divide the food's fat calories by its total calories and then multiply the result by 100. The fat percentage should be displayed with zero decimal places. The program should display an appropriate error message if either or both input values are less than zero.

Figure 5-28 Problem specification for Lab 5-2

First, analyze the problem, looking for the output first and then for the input. In this case, the user wants the program to display a food's fat calories and its fat percentage. To calculate these values, the computer will need to know the food's total calories and its grams of fat; both of these items will be entered by the user. Next, plan the algorithm. As you know, most algorithms begin with an instruction to enter the input items into the computer, followed by instructions that process the input items, typically including the items in one or more calculations. Most algorithms end with one or more instructions that display, print, or store the output items. Figure 5-29 shows the completed IPO chart for the health club problem.

Input	Processing	Output
total calories grams of fat	Processing items: none	fat calories fat percentage (0 decimal places)

```
Algorithm:
1. enter the total calories and grams of fat
2. if (both the total calories and grams of
      fat are greater than or equal to zero)
      calculate fat calories by multiplying
      grams of fat by 9
      calculate fat percentage by dividing
      fat calories by total calories and then
      multiplying the result by 100
      display the fat calories and fat
      percentage
   else
      display an error message
   end if
```

Figure 5-29 Completed IPO chart for the health club problem

After completing the IPO chart, you then move on to the third step in the problem-solving process, which is to desk-check the algorithm. You will desk-check the health club algorithm three times. For the first desk-check, you will use 150 as the total calories and 6 as the grams of fat; the fat calories and fat percentage should be 54 and 36%, respectively. For the second desk-check, you will use 105 as the total calories and 2 as the grams of fat. Using this data, the fat calories and fat percentage should be 18 and 17%, respectively. For the third desk-check, you will use 100 as the total calories and –3 as the grams of fat. Using this data, the program should display an error message. Figure 5-30 shows the completed desk-check table. Notice that the amounts in the fat calories and fat percentage columns agree with the results of the manual calculations.

total calories	grams of fat	fat calories	fat percentage
~~150~~	~~6~~	~~54~~	~~36~~
~~105~~	~~2~~	18	17
100	-3		

Figure 5-30 Completed desk-check table for the health club algorithm

The fourth step in the problem-solving process is to code the algorithm into a program. You begin by declaring memory locations that will store the values of the input, processing (if any), and output items. The health club problem will need four memory locations to store the input and output items. The input items (total calories and grams of fat) will be stored in variables, because the user should be allowed to change the value of those items during runtime. The output items (fat calories and fat percentage) also will be stored in variables, because their values are based on the current values of the input items. The total calories, grams of fat, and fat calories will

be integers; therefore, you will store those values in `int` variables. You will store the fat percentage in a `double` variable, because its value will be a real number. Figure 5-31 shows the input, processing, and output items from the IPO chart, along with the corresponding C++ instructions. Pay particular attention to the shaded statement, which calculates the fat percentage. The statement uses the `static_cast` operator to convert the integers stored in the `fatCals` and `totalCals` variables to `double` numbers before performing the division operation. As you learned in Chapter 4, the `static_cast` operator ensures that the quotient obtained when dividing one `int` variable by another `int` variable is a real number rather than an integer. Although the statement converts the contents of both `int` variables to `double`, recall that only one of the two integers involved in a division operation needs to be converted. As a result, you also can write the division part of the statement as either `static_cast<double>(fatCals) / totalCals` or `fatCals / static_cast<double>(totalCals)`.

IPO chart information	C++ instructions
Input	
total calories	`int totalCals = 0;`
grams of fat	`int fatGrams = 0;`
Processing	
none	
Output	
fat calories	`int fatCals = 0;`
fat percentage (0 decimal places)	`double fatPercent = 0.0;`
Algorithm	

1. enter the total calories and grams of fat	`cout << "Total calories: ";`
	`cin >> totalCals;`
	`cout << "Grams of fat: ";`
	`cin >> fatGrams;`

2. if (both the total calories and grams of fat are greater than or equal to zero)	`if (totalCals >= 0 && fatGrams >= 0)` `{`
calculate fat calories by multiplying grams of fat by 9	`fatCals = fatGrams * 9;`

calculate fat percentage by dividing fat calories by total calories and then multiplying the result by 100	`fatPercent =`
	`static_cast<double>(fatCals) /`
	`static_cast<double>(totalCals)`
	`* 100;`

display fat calories and fat percentage	`cout << "Fat calories: " <<`
	`fatCals << endl;`
	`cout << fixed << setprecision(0);`
	`cout << "Fat percentage: " <<`
	`fatPercent << "%" << endl;`

	`}`
else	`else`
display an error message	`cout << "Input error" << endl;`
end if	`//end if`

Figure 5-31 IPO chart information and C++ instructions for the health club problem

The fifth step in the problem-solving process is to desk-check the program. You begin by placing the names of the declared variables and named constants (if any) in a new desk-check table, along with their initial values. You then desk-check the remaining C++ instructions in order, recording in the desk-check table any changes made to the variables. Figure 5-32 shows the completed desk-check table for the program. The results agree with those shown in the algorithm's desk-check table in Figure 5-30.

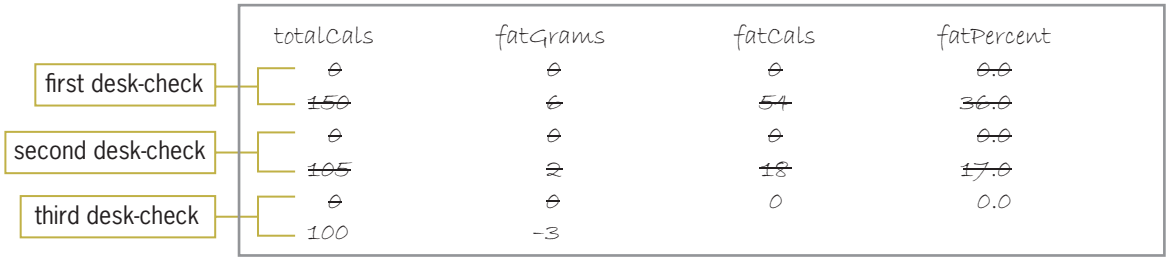


Figure 5-32 Completed desk-check table for the health club program

The final step in the problem-solving process is to evaluate and modify (if necessary) the program. Recall that you evaluate a program by entering its instructions into the computer and then using the computer to run (execute) it. While the program is running, you enter the same sample data used when desk-checking the program.

DIRECTIONS

Follow the instructions for starting your C++ development tool. Depending on the development tool you are using, you may need to create a new project; if so, name the project Lab5-2 Project and save it in the Cpp6\Chap05 folder. Enter the instructions shown in Figure 5-33 in a source file named Lab5-2.cpp. (Do not enter the line numbers.) Save the file in either the project folder or the Cpp6\Chap05 folder. Now, follow the appropriate instructions for running the Lab5-2.cpp file. Use the sample data from Figure 5-32 to test the program. If necessary, correct any bugs (errors) in the program.

```
 1  //Lab5-2.cpp - displays a food's fat calories and fat percentage
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <iomanip>
 6  using namespace std;
 7
 8  int main()
 9  {
10     //declare variables
11     int totalCals     = 0;
12     int fatGrams      = 0;
13     int fatCals       = 0;
14     double fatPercent = 0.0;
15
```

Figure 5-33 Health club program *(continues)*

(continued)

```
16      //enter input items
17      cout << "Total calories: ";
18      cin >> totalCals;
19      cout << "Grams of fat: ";
20      cin >> fatGrams;
21
22      //determine whether the data is valid
23      if (totalCals >= 0 && fatGrams >= 0)
24      {
25           //calculate and display the output
26           fatCals = fatGrams * 9;
27           fatPercent = static_cast<double>(fatCals)
28                / static_cast<double>(totalCals) * 100;
29
30           cout << "Fat calories: " << fatCals << endl;
31           cout << fixed << setprecision(0);
32           cout << "Fat percentage: " << fatPercent << "%" << endl;
33      }
34      else
35           cout << "Input error" << endl;
36      //end if
37
38      system("pause");
39      return 0;
40  }  //end of main function
```

If your C++ development tool does not require this statement, either omit it or make it a comment.

Figure 5-33 Health club program



LAB 5-3 Modify

If necessary, create a new project named Lab5-3 Project. Enter (or copy) the Lab5-2.cpp instructions into a new source file named Lab5-3.cpp. Currently, the `if` statement's true path handles valid data, while its false path handles invalid data. Modify the `if` statement so that invalid data is handled in the true path and valid data is handled in the false path. Also be sure to change Lab5-2.cpp in the first comment to Lab5-3.cpp. Use the sample data from Figure 5-32 to test the program.



LAB 5-4 Desk-Check

Desk-check the code shown in Figure 5-34 using the letter P. Although the code displays the appropriate message, it is considered inefficient. Why? How can you fix the code to make it more efficient?

```
//declare variable
char letter = ' ';

//enter input item, then display message
cout << "Enter a letter: ";
cin >> letter;

if (letter == 'P' || letter == 'p')
    cout << "Pass" << endl;
//end if
if (letter != 'P' || letter != 'p')
    cout << "Fail" << endl;
//end if
```

Figure 5-34 Code for Lab 5-4



LAB 5-5 Debug

Follow the instructions for starting C++ and opening the Lab5-5.cpp file. Test the program using codes of 1, 2, and 3. Debug the program.

Summary

- You use the selection structure when you want a program to make a decision before selecting the next instruction to process.
- Studying the problem specification will help you determine whether a solution requires a selection structure.
- A selection structure's condition must evaluate to either true or false. In both single-alternative and dual-alternative selection structures, the special instructions to follow when the structure's condition is true are placed in the structure's true path. In a dual-alternative selection structure, the special instructions to follow when the structure's condition is false are placed in the structure's false path. You should indent the instructions in both paths.
- A diamond is used to represent a selection structure's condition in a flowchart. The diamond is called the decision symbol. Each selection structure diamond has one flowline entering the symbol and two flowlines leaving the symbol. One of the flowlines leading out of a diamond should be marked with a "T" (for true) and the other should be marked with an "F" (for false).
- In most C++ programs, you will use the `if` statement to code a selection structure. The statement's condition must evaluate to either true or false.
- If either an `if` statement's true path or its false path contains more than one statement, the statements in the path must be entered as a statement block, which means the statements must be enclosed in a set of braces (`{}`).

- It is a good programming practice to include a comment (such as `//end if`) to identify the end of an `if` statement in a program.
- You use comparison operators to compare values in expressions. Expressions containing comparison operators always evaluate to either true or false. If more than one comparison operator with the same precedence number appears in a C++ expression, the computer evaluates those operators from left to right in the expression.
- You shouldn't use either the equality operator (==) or the inequality operator (!=) to compare two real numbers, because not all real numbers can be stored precisely in memory.
- A memory location declared in an `if` statement's true path can be used only by the instructions following the declaration statement within the true path. Likewise, a memory location declared in an `if` statement's false path can be used only by the instructions following the declaration statement within the false path.
- The And and Or logical operators are represented in C++ by the symbols && and ||, respectively. All expressions containing a logical operator evaluate to either true or false.
- In an expression, arithmetic operators are evaluated first, followed by comparison operators, followed by logical operators.
- Character comparisons in C++ are case sensitive.
- C++ provides the `toupper` and `tolower` built-in functions for temporarily converting a character to uppercase and lowercase, respectively.
- C++ provides the `fixed` and `scientific` stream manipulators for formatting the display of real numbers. It provides the `setprecision` stream manipulator for controlling the number of decimal places that appear when a real number is displayed. The `fixed` and `scientific` stream manipulators are defined in the iostream file. The `setprecision` stream manipulator is defined in the iomanip file.

Key Terms

Argument—an item that appears between the parentheses that follow a function's name; represents information that the function needs to perform its task

Boolean operators—another term for logical operators

Comparison operators—operators used to compare values having the same data type in an expression; also called relational operators; `<, <=, >, >=, ==, !=`

Data validation—the process of verifying that a program's input data is within the expected range

Decision symbol—the diamond in a flowchart; used to represent the condition in either a selection or repetition structure

Dual-alternative selection structures—selection structures that contain instructions in both their true and false paths

False path—contains the instructions to be processed when a dual-alternative selection structure's condition evaluates to false

fixed stream manipulator—the manipulator used to display real numbers in fixed-point notation

Local variable—a variable declared within a statement block; can be used only by the instructions within the statement block in which it is declared, and the instructions must appear after the declaration statement

Logical operators—operators used to combine two or more sub-conditions into one compound condition; also called Boolean operators

`scientific` stream manipulator—the manipulator used to display real numbers in scientific (e) notation

Selection structure—one of the three control structures; tells the computer to make a decision before selecting the next instruction to process; also called the decision structure

`setprecision` stream manipulator—the manipulator used to control the number of decimal places that appear when a real number is displayed

Short-circuit evaluation—refers to the way the computer evaluates two sub-conditions connected by a logical operator; when the logical operator is And, the computer does not evaluate sub-condition2 when sub-condition1 is false; when the logical operator is Or, the computer does not evaluate sub-condition2 when sub-condition1 is true

Single-alternative selection structure—a selection structure that requires a special action to be taken only when the structure's condition is true

Statement block—one or more instructions enclosed in a set of braces (`{}`)

`tolower` function—temporarily converts a character to lowercase

`toupper` function—temporarily converts a character to uppercase

True path—contains the instructions to be processed when a selection structure's condition evaluates to true

Truth tables—tables that summarize how the computer evaluates the logical operators in an expression

Review Questions

1. If an `if` statement's false path contains the statement `int sum = 0;`, where can the `sum` variable be used?
   a. in any instruction after the declaration statement in the entire program
   b. in any instruction after the declaration statement in the `if` statement
   c. in any instruction after the declaration statement in the `if` statement's false path
   d. none of the above, because you can't declare a variable in an `if` statement's false path

2. Which of the following is a valid `if` clause? (The `sales` variable has the `double` data type.)

   a. `if (sales > 500.0 && < 800.0)`

   b. `if (sales > 500.0 || < 800.0)`

   c. `if (sales > 500.0 And sales < 800.0)`

   d. `if (sales > 500.0 && sales < 800.0)`

3. Which of the following conditions results in true when the `initial` variable contains the letter A in either uppercase or lowercase?

   a. `if (initial = 'A' || initial = 'a')`

   b. `if (initial == 'A' || initial == 'a')`

   c. `if (initial = 'A' || initial = 'a')`

   d. `if (initial == 'A' && initial = 'a')`

4. The expression `3 < 6 && 7 < 4` evaluates to _____.

   a. true

   b. false

5. The computer will perform short-circuit evaluation when processing which of the following `if` clauses?

   a. `if (3 * 2 < 4 && 5 > 3)`

   b. `if (6 < 9 || 5 > 3)`

   c. `if (12 > 4 * 4 && 6 > 2)`

   d. all of the above

6. If an expression does not contain any parentheses, which of the following operators is performed first in the expression?

   a. arithmetic

   b. comparison

   c. logical

   d. you can't tell without seeing the expression

7. The expression `4 * 3 < 6 + 7 && 7 < 6 + 9` evaluates to _____.

   a. true

   b. false

8. Which of the following compares the contents of an `int` variable named `quantity` with the number 5?

   a. `if (quantity = 5)`

   b. `if (quantity == 5)`

   c. `if (quantity ≠ 5)`

   d. `if (quantity =! 5)`

9. Which of the following is required in a program that uses the `setprecision` stream manipulator?

   a. `#include <iostream>`

   b. `#include <setprecision>`

   c. `#include <iomanip>`

   d. `#include <manipulators>`

10. Which of the following tells the computer to display real numbers in fixed-point notation with no decimal places?

    a. `cout << fixed << decimal(0);`

    b. `cout << fixed << precision(0);`

    c. `cout << fixed << setprecision(0);`

    d. `cout << fixed << setdecimal(0);`

Exercises



Pencil and Paper

TRY THIS

1. Write the C++ code to compare the contents of an `int` variable named `quantity` with the number 10. If the variable's value is equal to 10, display the "Equal" message; otherwise, display the "Not equal" message. (The answers to TRY THIS Exercises are located at the end of the chapter.)

TRY THIS

2. Write the C++ code that corresponds to the partial flowchart shown in Figure 5-35. Use `int` variables named `sold` and `bonus`. (The answers to TRY THIS Exercises are located at the end of the chapter.)

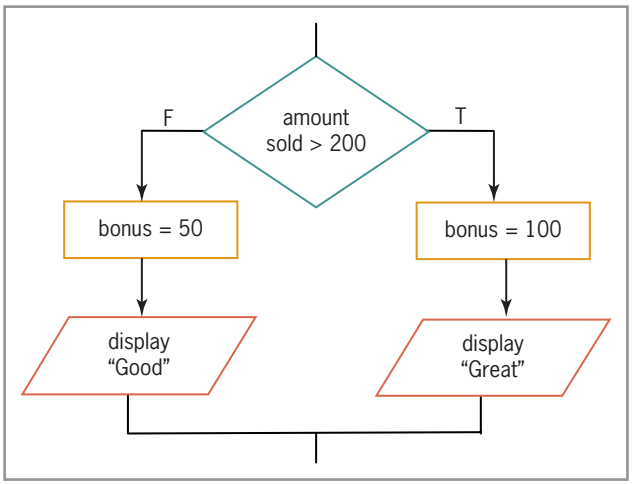


Figure 5-35

3. Complete TRY THIS Exercise 1, and then modify the code so that the true path displays "Not equal" and the false path displays "Equal". MODIFY THIS

4. Write the C++ code to display the message "Entry error" when the contents of the `units` variable is less than or equal to 0. Otherwise, calculate the total owed by multiplying the `units` variable's value by 5. Store the total owed in the `total` variable, and then display the total owed. INTRODUCTORY

5. A program stores sales amounts in two `double` variables named `marySales` and `jimSales`. Write the C++ code to assign the highest and lowest sales amounts to the `highSales` and `lowSales` variables, respectively, and then display the contents of those variables. (You can assume that both sales amounts are different.) INTERMEDIATE

6. A program uses a `char` variable named `department` and two `double` variables named `salary` and `raise`. The `department` variable contains one of the following letters (entered in either uppercase or lowercase): A, B, or C. Employees in departments A and B are receiving a 2% raise. Employees in department C are receiving a 1.5% raise. Write the C++ code to calculate and display the appropriate raise amount. Display the raise amount in fixed-point notation with two decimal places. ADVANCED

7. Correct the errors in the lines of code shown in Figure 5-36. The `code` variable has the `char` data type. SWAT THE BUGS

```
if (toupper(code) = 'x')
    cout << "Discountinued" << endl;
else
    cout << "Available" << endl;
    cout << "The item will be shipped ASAP" << endl;
//end if
```

Figure 5-36

Computer

TRY THIS

8. Code the flowchart shown earlier in Figure 5-5. Enter the C++ instructions into a source file named TryThis8.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Display the total owed in fixed-point notation with two decimal places. Save and run the program. Test the program using $2.25 and 6 as the price and quantity, respectively. The answer should be $12.15. Now test it using $2.25 and 5. (The answers to TRY THIS Exercises are located at the end of the chapter.)

TRY THIS

9. Complete Figure 5-37 by writing the algorithm and corresponding C++ instructions. Employees with a pay code of either 4 or 9 receive a 5% raise; all other employees receive a 3% raise. Enter the C++ instructions into a source file named TryThis9.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Display the new pay in fixed-point notation with no decimal places. Save and run the program. Test the program using 1 and $500 as the pay code and current pay. The new pay should be $515. Now test the program using the following three sets of input values: 4 and 450, 9 and 500, 2 and 625. (The answers to TRY THIS Exercises are located at the end of the chapter.)

IPO chart information	**C++ instructions**
Input	
pay code	`char code = ' ';`
current pay	`double currentPay = 0.0;`
raise rate 1 (3%)	`const double RATE1 = .03;`
raise rate 2 (5%)	`const double RATE2 = .05;`
Processing	
raise	`double raise = 0.0;`
Output	
new pay	`double newPay = 0.0;`
Algorithm	

Figure 5-37

MODIFY THIS

10. Complete TRY THIS Exercise 9. Enter (or copy) the instructions from the TryThis9.cpp file into a new source file named ModifyThis10.cpp. Modify the code in the ModifyThis10.cpp file so that a 5% raise is also given to employees with pay codes of 2 and 6. Save and run the program. Test the program using 1 and $500 as the pay code and current pay. Now test it using the following four sets of input values: 4 and 450, 9 and 500, 2 and 625, 6 and 150.

11. Tea Time Company wants a program that allows a clerk to enter the number of pounds of tea ordered, the price per pound, and whether the customer should be charged a $15 shipping fee. The program should calculate and display the total amount the customer owes. Use an `int` variable for the number of pounds, a `double` variable for the price per pound, and a `char` variable for the shipping information.

INTRODUCTORY

    a. Create an IPO chart for the problem, and then desk-check the algorithm twice. For the first desk-check, use 10 as the number of pounds and 12.54 as the price per pound; the customer should be charged the shipping fee. For the second desk-check, use 5 as the number of pounds and 11.59 as the price; the customer should not be charged the shipping fee.

    b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 5-37. Then code the algorithm into a program.

    c. Desk-check the program using the same data used to desk-check the algorithm.

    d. Enter your C++ instructions into a source file named Introductory11.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Display the total amount owed in fixed-point notation with two decimal places.

    e. Save and run the program. Test the program using the same data used to desk-check the program.

12. Marcy's Department store is having a BoGoHo (Buy One, Get One Half Off) sale. The store manager wants a program that allows the salesclerk to enter the prices of two items. The program should both calculate and display the total amount the customer owes. The half-off should always be taken on the item having the lowest price. For example, if the items cost $24.99 and $10, the half-off would be taken on the $10 item.

INTRODUCTORY

    a. Create an IPO chart for the problem, and then desk-check the algorithm twice. For the first desk-check, use 24.99 and 10 as the prices. For the second desk-check, use 11.50 and 30.99.

    b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 5-37. Then code the algorithm into a program.

    c. Desk-check the program using the same data used to desk-check the algorithm.

    d. Enter your C++ instructions into a source file named Introductory12.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Display the total amount owed in fixed-point notation with two decimal places.

    e. Save and run the program. Test the program using the same data used to desk-check the program.

INTERMEDIATE

13. Allenton Water Department wants a program that calculates a customer's monthly water bill. The clerk will enter the current and previous meter readings. The program should calculate and display the number of gallons of water used and the total charge for the water. The charge for water is $7 per 1000 gallons. However, there is a minimum charge of $16.67. (In other words, every customer must pay at least $16.67.)

    a. Create an IPO chart for the problem, and then desk-check the algorithm twice. For the first desk-check, use 13000 and 16000 as the previous and current meter readings. For the second desk-check, use 1650 and 3675.

    b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 5-37. Then code the algorithm into a program.

    c. Desk-check the program using the same data used to desk-check the algorithm.

    d. Enter your C++ instructions into a source file named Intermediate13.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Display the total charge in fixed-point notation with two decimal places.

    e. Save and run the program. Test the program using the same data used to desk-check the program.

ADVANCED

14. A third-grade teacher at Plano Elementary School wants a program that allows a student to enter the amount of money a customer owes and the amount of money the customer paid. The program should calculate and display the amount of change, as well as how many dollars, quarters, dimes, nickels, and pennies to return to the customer. Display an appropriate message when the amount paid is less than the amount owed.

    a. Create an IPO chart for the problem, and then desk-check the algorithm three times. For the first desk-check, use 75.34 and 80 as the amount owed and paid, respectively. For the second desk-check, use 39.67 and 50. For the third desk-check, use 10.55 and 9.75.

    b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 5-37. Then code the algorithm into a program.

    c. Desk-check the program using the same data used to desk-check the algorithm.

    d. Enter your C++ instructions into a source file named Advanced14.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Display the change in fixed-point notation with two decimal places. Display the remaining output in fixed-point notation with no decimal places.

    e. Save and run the program. Test the program using the same data used to desk-check the program.

ADVANCED

15. As you learned in the chapter, you must be careful when comparing two real numbers for either equality or inequality, because some real numbers cannot be stored precisely in memory. To determine whether two real numbers are either equal or unequal, you should test that the difference between both numbers is less than some acceptable small value, such as .00001.

    a. Follow the instructions for starting C++ and opening the Advanced15.cpp file. Notice that the code divides the contents of the `num1` variable (10.0) by the contents of the `num2` variable (3.0), storing the result (approximately 3.33333) in the `quotient` variable. An `if` statement is used to compare the contents of the `quotient` variable with the number 3.33333. The `if` statement displays a message that indicates whether the numbers are equal.

    b. If necessary, make the `system("pause");` statement a comment, and then save the program. Run the program. Even though the message on the screen states that the quotient is 3.33333, the message indicates that this value is not equal to 3.33333. Stop the program.

    c. If you need to compare two real numbers for equality or inequality, first find the difference between both numbers and then compare the absolute value of that difference to a small number, such as .00001. You can use the C++ `fabs` function to find the absolute value of a real number. The absolute value of a number is a positive number that represents the distance the number is from 0 on the number line. For example, the absolute value of the number 5 is 5. The absolute value of the number -5 also is 5. To use the `fabs` function, the program must contain the `#include <cmath>` directive. Modify the program appropriately. Save and then run the program. This time, the message "Yes, the quotient 3.33333 is equal to 3.33333." appears. Stop the program.

SWAT THE BUGS

16. Follow the instructions for starting C++ and opening the SwatTheBugs16.cpp file. If necessary, make the `system("pause");` statement a comment, and then save the program. Run and then debug the program.

Answers to TRY THIS Exercises



Pencil and Paper

1. See Figure 5-38.

```
if (quantity == 10)
    cout << "Equal" << endl;
else
    cout << "Not equal" << endl;
//end if
```

Figure 5-38

2. See Figure 5-39.

```
if (sold > 200)
{
    bonus = 100;
    cout << "Great" << endl;
}
else
{
    bonus = 50;
    cout << "Good" << endl;
}   //end if
```

Figure 5-39



Computer

8. See Figures 5-40.

```
 1  //TryThis8.cpp - displays the total owed
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <iomanip>
 6  using namespace std;
 7
 8  int main()
 9  {
10     //declare variables
11     int quantity    = 0;
12     double price    = 0.0;
13     double discount = 0.0;
14     double total    = 0.0;
15
16     cout << "Price: ";
17     cin >> price;
18     cout << "Quantity: ";
19     cin >> quantity;
20
21     //calculate total owed
22     total = price * quantity;
23
24     // calcualte discount, if necessary
25     if (quantity > 5)
26     {
27         discount = total * .1;
28         total = total - discount;
29     }   //end if
30
31     //display total owed
32     cout << fixed << setprecision(2);
33     cout << "Total owed: $" << total << endl;
34
35     system("pause");
36     return 0;
37  }  //end of main function
```

your C++ development tool may not require this statement

Figure 5-40

9. See Figures 5-41 and 5-42.

IPO chart information	C++ instructions
Input	
pay code	`char code = ' ';`
current pay	`double currentPay = 0.0;`
raise rate 1 (3%)	`const double RATE1 = .03;`
raise rate 2 (5%)	`const double RATE2 = .05;`
Processing	
raise	`double raise = 0.0;`
Output	
new pay	`double newPay = 0.0;`
Algorithm	

1. enter the pay code and current pay	`cout << "Pay code: ";`
	`cin >> code;`
	`cout << "Current pay: ";`
	`cin >> currentPay;`

2. if (the pay code is 4 or 9)	`if (code == '4' || code == '9')`
calculate the raise by multiplying the current pay by raise rate 2	`raise = currentPay * RATE2;`
else	`else`
calculate the raise by multiplying the current pay by raise rate 1	`raise = currentPay * RATE1;`
end if	`//end if`
3. calculate the new pay by adding the raise to the current pay	`newPay = currentPay + raise;`
4. display the new pay	`cout << "New pay: $" <<` `newPay << endl;`

Figure 5-41

```
 1  //TryThis9.cpp - displays the new pay
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <iomanip>
 6  using namespace std;
 7
 8  int main()
 9  {
10     //declare named constants and variables
11     const double RATE1 = .03;
12     const double RATE2 = .05;
13     char code          = ' ';
14     double currentPay  = 0.0;
15     double raise       = 0.0;
16     double newPay      = 0.0;
17
18     //enter input items
19     cout << "Pay code: ";
20     cin >> code;
21     cout << "Current pay: ";
22     cin >> currentPay;
23
24     //calculate raise and new pay
25     if (code == '4' || code == '9')
26        raise = currentPay * RATE2;
27     else
28        raise = currentPay * RATE1;
29     //end if
30     newPay = currentPay + raise;
31
32     //display new pay
33     cout << fixed << setprecision(0);
34     cout << "New pay: $" << newPay << endl;
35
36     system("pause");
37     return 0;
38  }  //end of main function
```

your C++ development tool may not require this statement

Figure 5-42

CHAPTER 6

More on the Selection Structure

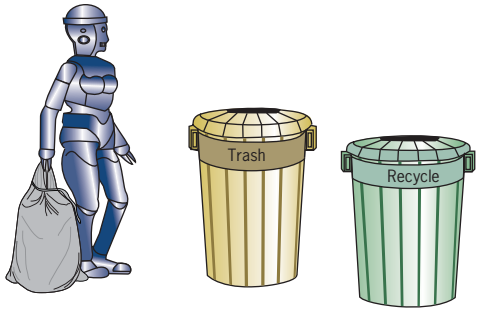
After studying Chapter 6, you should be able to:

- Include a nested selection structure in pseudocode and in a flowchart
- Code a nested selection structure
- Recognize common logic errors in selection structures
- Include a multiple-alternative selection structure in pseudocode and in a flowchart
- Code a multiple-alternative selection structure in C++

Making Decisions

In Chapter 5, you learned that you use the selection structure when you want the computer to make a decision and then select the appropriate path—either the true path or the false path—based on the result. Both paths in a selection structure can include instructions that declare variables, perform calculations, and so on. In this chapter, you will learn that both paths also can include other selection structures. When either a selection structure's true path or its false path contains another selection structure, the inner selection structure is referred to as a **nested selection structure**, because it is contained (nested) within the outer selection structure. Similar to the initial examples of selection structures in Chapter 5, the first examples of nested selection structures will involve Robin, the mechanical woman. The first problem specification and its algorithm are shown in Figure 6-1. (The problem specification and algorithm, which is written in pseudocode, are from Figure 5-4 in Chapter 5.) The algorithm requires a selection structure, but not a nested one. This is because only one decision—whether Robin is holding a bag of trash—is necessary.

Robin is holding either a bag of trash or a bag of recyclables in her right hand. She is directly in front of two containers: one marked Trash and the other marked Recycle. A lid is on each container. Robin needs to lift the lid from the appropriate container, then drop the bag in the container, and then put the lid back on the container.



condition

```
if (the bag contains trash)
    lift the Trash container's lid using your left hand
    drop the bag of trash in the Trash container
    put the lid back on the Trash container using your left hand
else
    lift the Recycle container's lid using your left hand
    drop the bag of recyclables in the Recycle container
    put the lid back on the Recycle container using your left hand
end if
```

true path

false path

Figure 6-1 A problem that requires a dual-alternative selection structure

Now let's make a slight change to the problem specification in Figure 6-1. This time, the status of the lid on each container is not known: one or both of the lids could be on or off. Consider the changes you will need to make to the original algorithm in Figure 6-1. The first instruction in the original algorithm represents the selection structure's condition. The condition requires Robin to make a decision about the contents of the bag she is holding; Robin still will need to make this decision. The next three instructions tell Robin what to do when the condition is true, which is when the bag contains trash. The first instruction in the true path directs Robin to lift the Trash container's lid. That instruction was correct for the original problem specification, which states that the lid is on the container. However, in the modified problem specification, the status of the lid is not known. Therefore, Robin first will need to make a decision about the lid's status and then only lift the lid if it's on the container. The last two instructions in the true path direct Robin to drop the bag of trash in the Trash container and then put the lid back on the container; Robin still will need to follow both instructions. Now look at the false path of the selection structure in Figure 6-1. The first instruction in the false path directs Robin to lift the lid from the Recycle container. Here again, Robin first will need to determine whether it's necessary to do this. The last two instructions in the false path tell Robin to drop the bag of recyclables in the Recycle container and then put the lid back on the container; Robin still will need to follow both instructions. Figure 6-2 shows the modified problem specification and algorithm. The modified algorithm contains an outer dual-alternative selection structure and two nested single-alternative selection structures. The outer selection structure begins with *if (the bag contains trash)*, and it ends with the last *end if*. The *else* belongs to the outer selection structure and separates the structure's true path from its false path. Notice that the instructions in both paths are indented within the outer selection structure. Indenting in this manner clearly indicates the instructions to be followed when Robin is holding a bag of trash, as well as the ones to be followed when the bag does not contain trash. One of the nested selection structures appears in the outer selection structure's true path, and the other appears in its false path. The nested selection structure in the true path begins with *if (the lid is on the Trash container)*, and it ends with the first *end if*. The instruction between both lines is indented to indicate that it is part of the nested selection structure. The nested selection structure in the false path begins with *if (the lid is on the Recycle container)*, and it ends with the second *end if*. Here again, the instruction between both lines is indented within the nested selection structure. For a nested selection structure to work correctly, it must be contained entirely within the outer selection structure. Each nested selection structure in Figure 6-2, for example, appears entirely within its corresponding path in the outer selection structure.

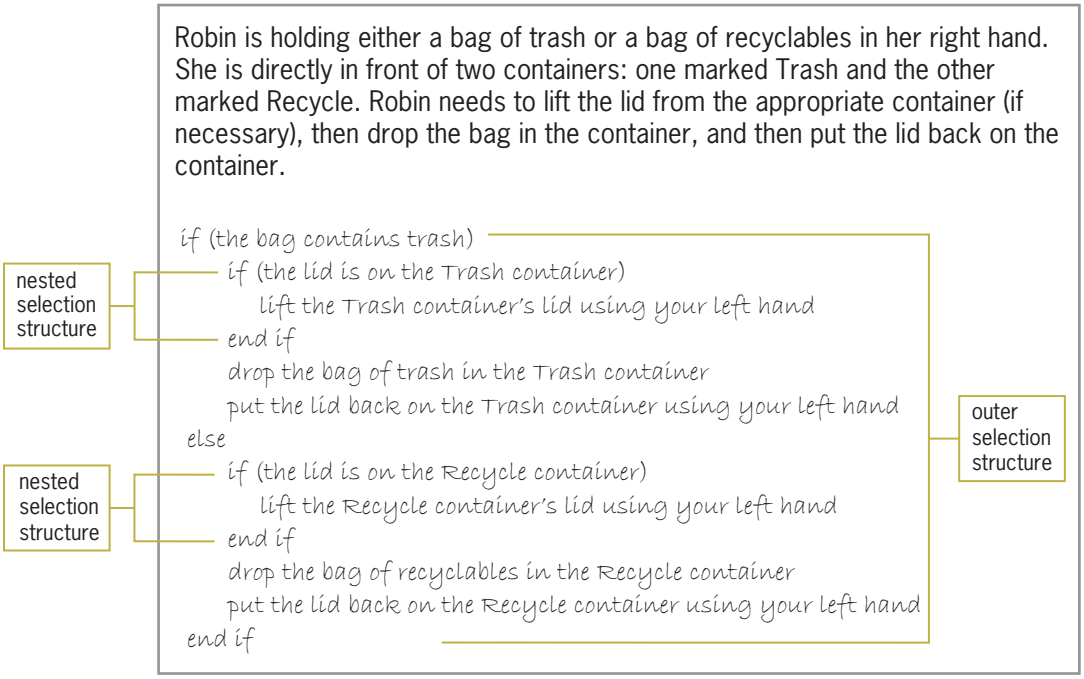

Robin is holding either a bag of trash or a bag of recyclables in her right hand. She is directly in front of two containers: one marked Trash and the other marked Recycle. Robin needs to lift the lid from the appropriate container (if necessary), then drop the bag in the container, and then put the lid back on the container.
if (the bag contains trash)
if (the lid is on the Trash container)
lift the Trash container's lid using your left hand
end if
drop the bag of trash in the Trash container
put the lid back on the Trash container using your left hand
else
if (the lid is on the Recycle container)
lift the Recycle container's lid using your left hand
end if
drop the bag of recyclables in the Recycle container
put the lid back on the Recycle container using your left hand
end if
nested selection structure
nested selection structure
outer selection structure

Figure 6-2 A problem that requires nested single-alternative selection structures

Figure 6-3 shows another problem specification and algorithm (also written in pseudocode) involving Robin. As the algorithm indicates, Robin needs to ask the store clerk whether the store accepts the Discovery card. Depending on the answer, Robin will use either her Discovery card or cash to pay for the items she is purchasing.

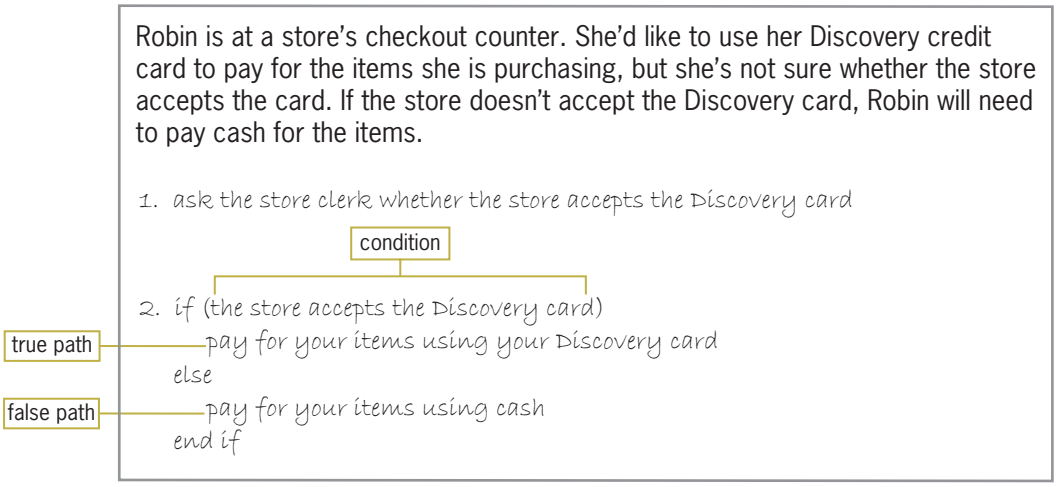

Robin is at a store's checkout counter. She'd like to use her Discovery credit card to pay for the items she is purchasing, but she's not sure whether the store accepts the card. If the store doesn't accept the Discovery card, Robin will need to pay cash for the items.
1. ask the store clerk whether the store accepts the Discovery card
condition
2. if (the store accepts the Discovery card)
pay for your items using your Discovery card
else
pay for your items using cash
end if
true path
false path

Figure 6-3 Another problem that requires a dual-alternative selection structure

Now let's change Figure 6-3's problem specification. This time, Robin would like to use either her Discovery card or her Vita card, but she prefers to use her Discovery card. If the store does not take either card, Robin will need to pay cash for the items she is purchasing. Consider the changes you will need

to make to the original algorithm in Figure 6-3. Robin still will need to ask the store clerk whether the store accepts the Discovery card, and if it does, she should use that card to pay for her items. Therefore, the first three lines in the original algorithm do not need to be changed. The next line in the algorithm is the *else* line. The modified algorithm still will need this line to indicate that there are tasks to be performed when the store does not accept the Discovery card. The next line in the original algorithm tells Robin to use cash to pay for her items. That instruction was correct for the original problem specification, which gave Robin only two payment choices: either her Discovery card or cash. However, in the modified problem specification, Robin has three payment choices: her Discovery card, her Vita card, or cash. Before paying with cash, Robin needs to inquire whether the store accepts the Vita card; if it does, Robin should use her Vita card to pay for her items. Robin should pay with cash only when the store does not accept either credit card. Figure 6-4 shows the modified problem specification and algorithm. The modified algorithm contains an outer dual-alternative selection structure and a nested dual-alternative selection structure. The outer selection structure begins with *if (the store accepts the Discovery card)*, and it ends with the last *end if*. The first *else* belongs to the outer selection structure and separates the structure's true path from its false path. Notice that the instructions in the outer selection structure's true and false paths are indented. The nested selection structure, which appears in the outer selection structure's false path, begins with *if (the store accepts the Vita card)*, and it ends with the first *end if*. The indented *else* belongs to the nested selection structure and separates the nested structure's true path from its false path. For clarity, the instructions in the nested selection structure's true and false paths are indented within the structure.

Robin is at a store's checkout counter. She'd like to use one of her credit cards—either her Discovery card or her Vita card, but preferably her Discovery card—to pay for the items she is purchasing. However, Robin is not sure whether the store accepts either card. If the store doesn't accept either card, Robin will need to pay cash for the items.

```
1. ask the store clerk whether the store accepts the Discovery card
2. if (the store accepts the Discovery card)
       pay for your items using your Discovery card
   else
       ask the store clerk whether the store accepts the Vita card
       if (the store accepts the Vita card)
           pay for your items using your Vita card
       else
           pay for your items using cash
       end if
   end if
```

nested selection structure

outer selection structure



Notice that the entire nested selection structure is contained in the outer selection structure's false path.

Figure 6-4 A problem that requires a nested dual-alternative selection structure

Flowcharting a Nested Selection Structure

Figure 6-5 shows a problem specification for a voter eligibility program. The program determines whether a person can vote and then displays one of three different messages. The appropriate message depends on the person's age and voter registration status. For example, if the person is younger than 18 years old, the program should display the message "You are too young to vote." However, if the person is at least 18 years old, the program should display one of two messages. The correct message to display is determined by the person's voter registration status. If the person is registered, then the appropriate message is "You can vote."; otherwise, it is "You must register before you can vote." Notice that determining the person's voter registration status is important only *after* his or her age is determined. Because of this, the decision regarding the age is considered the primary decision, while the decision regarding the registration status is considered the secondary decision, because whether it needs to be made depends on the result of the primary decision. A primary decision is always made by an outer selection structure, while a secondary decision is always made by a nested selection structure. Also included in Figure 6-5 is a correct solution to the voter eligibility problem in flowchart form. The first diamond in the flowchart represents the outer selection structure's condition, which checks whether the age entered by the user is greater than or equal to 18. If the condition evaluates to false, it means that the person is not old enough to vote. In that case, the outer selection structure's false path will display the "You are too young to vote." message before the outer selection structure ends. However, if the outer selection structure's condition evaluates to true, it means that the person *is* old enough to vote. Before displaying the appropriate message, the outer selection structure's true path gets the registration status from the user. It then uses a nested selection structure to determine whether the person is registered. The nested selection structure's condition is represented by the second diamond in Figure 6-5. If the person is registered, the nested selection structure's true path displays the "You can vote." message; otherwise, its false path displays the "You must register before you can vote." message. After the appropriate message is displayed, the outer and nested selection structures end. Notice that the nested selection structure is processed only when the outer selection structure's condition evaluates to true.

Problem specification
The Danville city manager wants a program that determines voter eligibility and displays one of three messages. The messages and the criteria for displaying each message are as follows:

Message	Criteria
You are too young to vote.	person is younger than 18 years old
You can vote.	person is at least 18 years old and is registered to vote
You must register before you can vote.	person is at least 18 years old but is not registered to vote

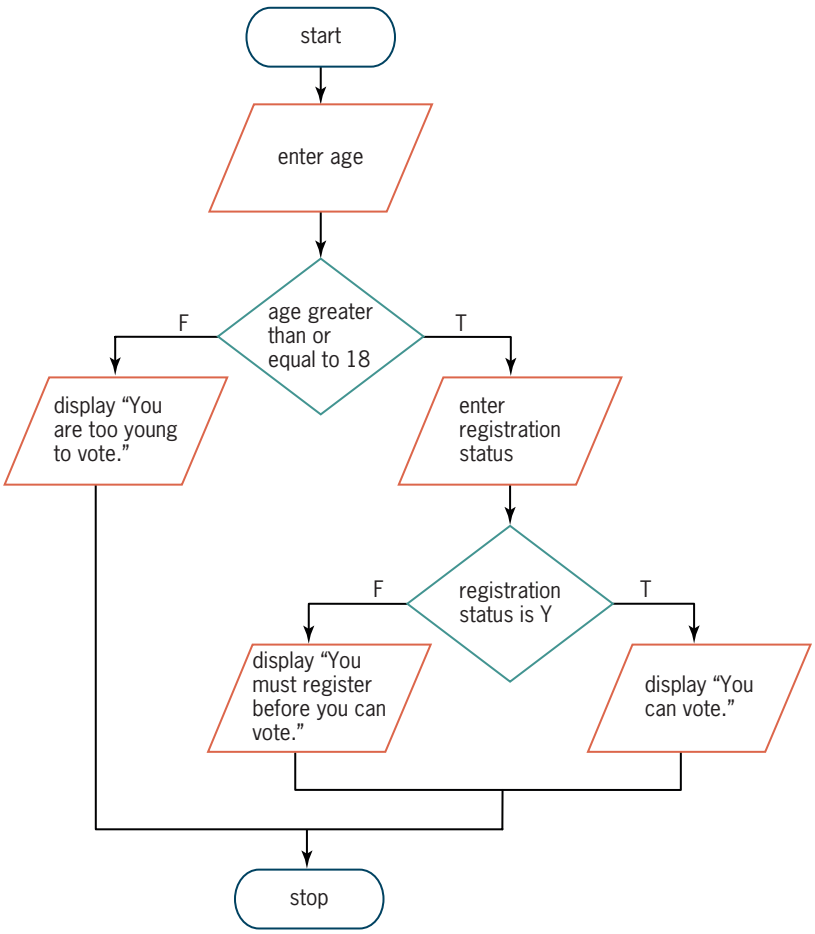


Figure 6-5 Problem specification and a correct solution for the voter eligibility problem

Figure 6-6 shows another correct solution, also in flowchart form, for the voter eligibility problem. As in the previous solution, the outer selection structure in this solution determines the age (the primary decision), and the nested selection structure determines the voter registration status (the secondary decision). In this solution, however, the outer selection structure's condition checks whether the age is less than 18. In addition, the nested selection structure appears in the outer selection structure's false path in this solution, which means it will be processed only when the outer selection structure's condition evaluates to false. The solutions in Figures 6-5 and 6-6 produce the same results. Neither solution is better than the other. Each simply represents a different way of solving the same problem.

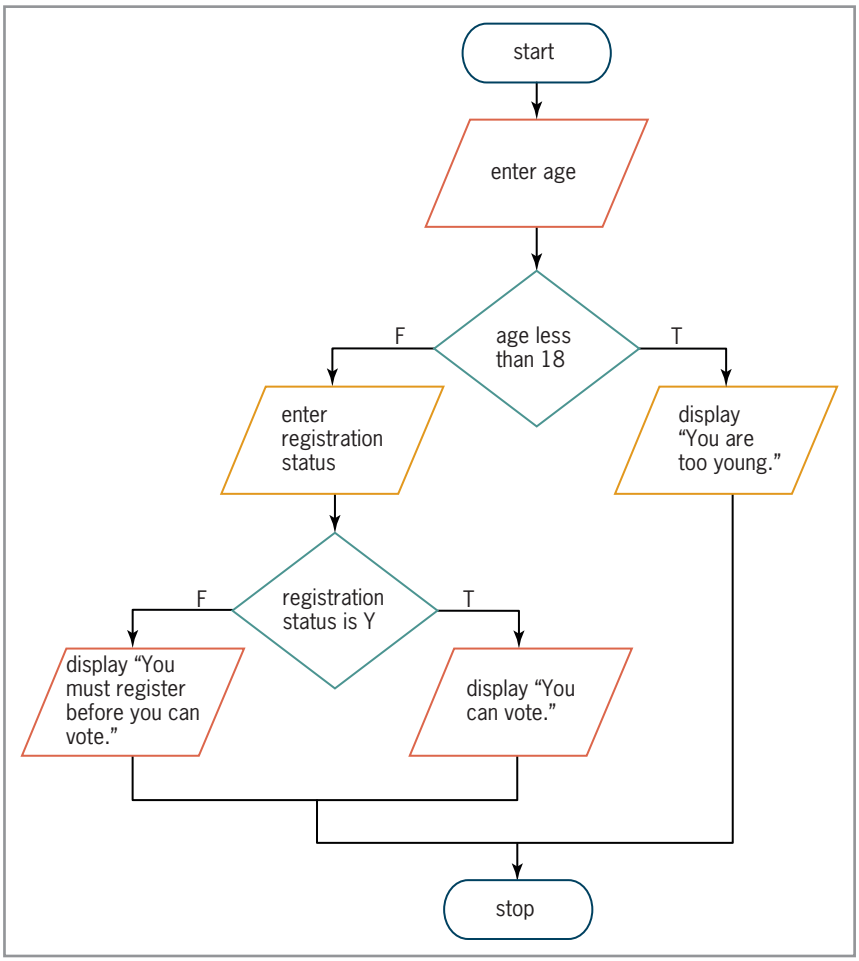


Figure 6-6 Another correct solution for the voter eligibility problem

Coding a Nested Selection Structure

In Chapter 5's Lab 5-2, you created a program for the manager of Willow Springs Health Club. The program allows the manager to enter the number of calories and grams of fat contained in a specific food. The program then calculates and displays two values: the food's fat calories and its fat percentage. The health club's manager now wants the program to display the message "High in fat" when the food's fat percentage is greater than 30%; otherwise, the program should display the message "Not high in fat". To accomplish this task, you will need to add a nested dual-alternative selection structure to the program. You will code the nested selection structure using the dual-alternative form of the `if` statement. The dual-alternative form is simply an `if` statement that contains an `else` clause. The modified problem specification and program are shown in Figure 6-7. The modifications made to the original specification and program are shaded in the figure. (The flowchart for the modified program is contained in the Ch6Flowcharts.pdf file, which is located in the Cpp\Chap06 folder.)

Problem specification
The manager of Willow Springs Health Club wants a program that allows her to enter the number of calories and grams of fat contained in a specific food. The program should calculate and display two values: the food's fat calories and its fat percentage. The fat calories are the number of calories attributed to fat. The fat percentage is the ratio of the food's fat calories to its total calories. You can calculate a food's fat calories by multiplying its fat grams by the number 9, because each gram of fat contains 9 calories. To calculate the fat percentage, you divide the food's fat calories by its total calories and then multiply the result by 100. The fat percentage should be displayed with zero decimal places. If the fat percentage is greater than 30%, the program should display the message "High in fat"; otherwise, it should display the message "Not high in fat".

```
 1  //Lab5-2.cpp - displays a food's fat calories and fat percentage
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <iomanip>
 6  using namespace std;
 7
 8  int main()
 9  {
10     //declare variables
11     int totalCals    = 0;
12     int fatGrams     = 0;
13     int fatCals      = 0;
14     double fatPercent = 0.0;
15
16     //enter input items
17     cout << "Total calories: ";
18     cin >> totalCals;
19     cout << "Grams of fat: ";
20     cin >> fatGrams;
21
22     //determine whether the data is valid
23     if (totalCals >= 0 && fatGrams >= 0)
24     {
25          //calculate and display the output
26          fatCals = fatGrams * 9;
27          fatPercent = static_cast<double>(fatCals)
28               / static_cast<double>(totalCals) * 100;
29
30          cout << "Fat calories: " << fatCals << endl;
31          cout << fixed << setprecision(0);
32          cout << "Fat percentage: " << fatPercent << "%" << endl;
33          if (fatPercent > 30.0)
34              cout << "High in fat" << endl;
35          else
36              cout << "Not high in fat" << endl;
37          //end if
38     }
```

nested selection structure

Figure 6-7 Modified problem specification and program for the health club problem from Chapter 5's Lab 5-2 *(continues)*

(continued)

```
39      else
40           cout << "Input error" << endl;
41      //end if
42
43      system("pause");
44      return 0;
45 }    //end of main function
```

Figure 6-7 Modified problem specification and program for the health club problem from Chapter 5's Lab 5-2

The outer selection structure's condition appears on Line 23 in the program and determines whether the user's input is valid. If the input is not valid, the outer selection structure's false path displays the "Input error" message. If the input is valid, on the other hand, the instructions in the outer selection structure's true path are processed. The true path begins on Line 24 and ends on Line 38. First, the true path calculates and then displays the food's fat calories and fat percentage. It then uses a nested selection structure to determine whether the fat percentage is greater than 30.0. The nested selection structure is on Lines 33 through 37 and is shaded in Figure 6-7. If the fat percentage is greater than 30.0, the nested selection structure's true path displays the message "High in fat"; otherwise, its false path displays the message "Not high in fat". Figure 6-8 shows a sample run of the modified health club program.

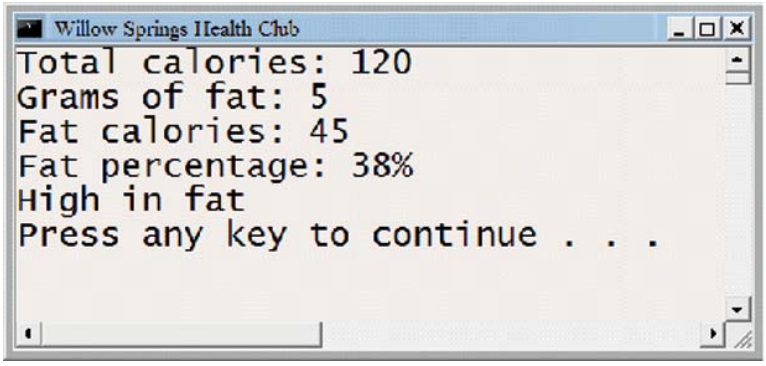

Willow Springs Health Club
Total calories: 120
Grams of fat: 5
Fat calories: 45
Fat percentage: 38%
High in fat
Press any key to continue . . .

Figure 6-8 Sample run of the modified health club program



The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 6-1

1. A selection structure should display the message "Great score!" when a student's test score is at least 90. When the test score is 70 through 89, the selection structure should display the message "Good score". For all other test scores, the selection structure should display the message "Retake the test". Write the pseudocode for the selection structure.

2. Draw a flowchart of the selection structure from Question 1.

3. Write the C++ code for the selection structure from either Question 1 or Question 2. The test score is stored in an `int` variable named `score`.

4. The manager of a golf course wants a program that displays the appropriate fee to charge a golfer. Club members pay a $5 fee. Non-members golfing on Monday through Thursday pay $15. Non-members golfing on Friday through Sunday pay $25. The condition in the program's outer selection structure should check the _____, while the condition in its nested selection structure should check the _____.

   a. membership status, day of the week

   b. day of the week, membership status

   c. membership status, fee

   d. fee, day of the week

Logic Errors in Selection Structures

In the next few sections, you will observe some of the common logic errors made when writing selection structures. Being aware of these errors will help prevent you from making them. In most cases, logic errors in selection structures are a result of one of the following three mistakes: using a compound condition rather than a nested selection structure, reversing the decisions in the outer and nested selection structures, or using an unnecessary nested selection structure. The XYZ Company's bonus program will be used to demonstrate these logic errors. The company pays each salesperson an 8% bonus on his or her sales. However, salespeople having a sales code of X receive an additional $150 bonus when their sales are greater than or equal to $10,000; otherwise, they receive an additional $125 bonus. Notice that the salesperson's code determines whether he or she receives an additional bonus. If the salesperson is entitled to the additional bonus, then the amount of his or her sales determines the appropriate additional amount. In this case, the decision regarding the salesperson's code is the primary decision, while the decision regarding the sales amount is the secondary decision. The pseudocode shown in Figure 6-9 represents a correct algorithm for the bonus program.

```
1. enter the code and sales
2. calculate the bonus by multiplying the sales by .08
3. if (the code is X)
        if (the sales are greater than or equal to 10000)
                add 150 to the bonus
        else
                add 125 to the bonus
        end if
   end if
4. display the bonus
```



You also can write the nested selection structure's condition as follows: `if (the sales are less than 10000)`. You then would reverse the instructions in the true and false paths.

Figure 6-9 A correct algorithm for the bonus problem

To verify its accuracy, you will desk-check the algorithm from Figure 6-9 three times. Figure 6-10 shows the values you will use for the desk-checks and also includes the manually calculated results.

Code	Sales	Bonus
X	$15000	$1350
X	$ 9000	$ 845
A	$13000	$1040

Figure 6-10 Test data and manually calculated results

The first instruction in the algorithm shown in Figure 6-9 is to enter the code and sales. For the first desk-check, the code and sales are X and 15000, respectively. The second instruction is to calculate the bonus by multiplying the sales by .08; the answer is 1200. Figure 6-11 shows the current status of the desk-check table.

code	sales	bonus
X	15000	1200

Figure 6-11 Current status of the desk-check table

Next, the outer selection structure in the algorithm determines whether the salesperson's code is X. It is, so the nested selection structure checks whether the sales are greater than or equal to 10000. The sales are greater than 10000, so the nested selection structure's true path adds 150 to the bonus; the answer is 1350. After the bonus is calculated, both selection structures end. The last instruction in the algorithm displays the bonus on the screen. Figure 6-12 shows the desk-check table after completing the first desk-check. The 1350 in the bonus column agrees with the manual calculation shown earlier in Figure 6-10.

code	sales	bonus
X	15000	~~1200~~
		1350

Figure 6-12 Desk-check table after completing the first desk-check

Using the second set of test data, the user enters X as the code and 9000 as the sales. The second instruction in the algorithm calculates the bonus by multiplying the sales by .08; the answer is 720. Next, the outer selection structure determines whether the salesperson's code is X. It is, so the nested selection structure checks whether the sales are greater than or equal to 10000. The sales are not greater than or equal to 10000, so the nested selection structure's false path adds 125 to the bonus amount; the answer is 845. After the bonus is calculated, both selection structures end. The last instruction in the algorithm displays the bonus on the screen. Figure 6-13 shows the desk-check table after completing the second desk-check. The 845 in the bonus column agrees with the manual calculation shown earlier in Figure 6-10.

code	sales	bonus
~~X~~	~~15000~~	~~1200~~
		~~1350~~
X	9000	~~720~~
		845

Figure 6-13 Desk-check table after completing the second desk-check

Using the third set of test data, the user enters A as the code and 13000 as the sales. The second instruction in the algorithm calculates the bonus by multiplying the sales by .08; the answer is 1040. Next, the outer selection structure determines whether the salesperson's code is X. The code is not X, so the outer selection structure ends. Notice that the nested selection structure is not processed when the outer selection structure's condition is false. The last instruction in the algorithm displays the bonus on the screen. Figure 6-14 shows the desk-check table after completing the third desk-check. The final entry in the bonus column (1040) agrees with the manual calculation shown earlier in Figure 6-10.

code	sales	bonus
~~X~~	~~15000~~	~~1200~~
		~~1350~~
~~X~~	~~9000~~	~~720~~
		~~845~~
A	13000	1040

Figure 6-14 Desk-check table after completing the third desk-check

First Logic Error: Using a Compound Condition Rather Than a Nested Selection Structure

A common error made when writing selection structures is to use a compound condition in the outer selection structure when a nested selection structure is needed. Figure 6-15 shows an example of this error in the bonus algorithm. The correct algorithm is included in the figure for comparison. Notice that the incorrect algorithm uses one selection structure rather than two selection structures and that the selection structure contains a compound condition. Consider why the selection structure in the incorrect algorithm cannot be used in place of the selection structures in the correct algorithm. In the correct algorithm, the outer and nested selection structures indicate that a hierarchy exists between the code and sales decisions: the code decision is always made first, followed by the sales decision (if necessary). In the incorrect algorithm, the compound condition indicates that no hierarchy exists between the code and sales decisions. Consider how this difference changes the algorithm.

Correct algorithm

```
1. enter the code and sales
2. calculate the bonus by multiplying
   the sales by .08
3. if (the code is X)
        if (the sales are greater than or
        equal to 10000)
              add 150 to the bonus
        else
              add 125 to the bonus
        end if
   end if
4. display the bonus
```

Incorrect algorithm

```
1. enter the code and sales
2. calculate the bonus by multiplying
   the sales by .08
3. if (the code is X and the sales are greater
   than or equal to 10000)
        add 150 to the bonus
   else
        add 125 to the bonus
   end if
4. display the bonus
```

uses a compound condition instead of a nested selection structure

Figure 6-15 Correct algorithm and an incorrect algorithm containing the first logic error

To understand why the incorrect algorithm in Figure 6-15 will not work correctly, you will desk-check it using the same test data used to desk-check the correct algorithm. The first instruction in the incorrect algorithm is to enter the code and sales. For the first desk-check, the code and sales are X and 15000, respectively. The second instruction is to calculate the bonus by multiplying the sales by .08; the answer is 1200. The compound condition in the third instruction determines whether the salesperson's code is X and, at the same time, the sales are greater than or equal to 10000. In this case, the compound condition evaluates to true. Therefore, the selection structure's true path adds 150 to the bonus, giving 1350, and then the selection structure ends. The last instruction in the incorrect algorithm displays the bonus (1350) on the screen. Even though its selection structure is phrased incorrectly, the incorrect algorithm produces the same result as the correct algorithm using the first set of test data.

Using the second set of test data, the user enters X as the code and 9000 as the sales. The second instruction in the incorrect algorithm multiplies the sales by .08, giving a bonus of 720. The compound condition in the third instruction determines whether the salesperson's code is X and, at the same time, the sales are greater than or equal to 10000. In this case, the compound condition evaluates to false, because the sales do not meet the specified criteria. As a result, the selection structure's false path adds 125 to the bonus, giving 845, and then the selection structure ends. The last instruction in the incorrect algorithm displays the bonus (845) on the screen. Here again, using the second set of test data, the incorrect algorithm produces the same result as the correct algorithm.

Using the third set of test data, the user enters A as the code and 13000 as the sales. The second instruction in the incorrect algorithm multiplies the sales by .08, giving a bonus of 1040. The compound condition in the third instruction determines whether the salesperson's code is X and, at the same time, the sales are greater than or equal to 10000. In this case, the compound condition evaluates to false, because the salesperson's code is not X. As a result, the selection structure's false path adds 125 to the bonus, giving 1165, and then the selection structure ends. The last instruction in the incorrect algorithm displays the bonus (1165) on the screen. Notice that the incorrect algorithm produces erroneous results using the third set of test data: According to Figure 6-10, the

correct bonus is 1040. It is important to desk-check an algorithm several times using different test data. In this case, if you had used only the first two sets of data to deck-check the incorrect algorithm, you would not have discovered the error. Figure 6-16 shows the desk-check table for the incorrect algorithm. As indicated in the figure, the results of the first and second desk-checks are correct, but the result of the third desk-check is not correct.

code	sales	bonus	
X	15000	1200	
		1350	(correct result for the first desk-check)
X	9000	720	
		845	(correct result for the second desk-check)
A	13000	1040	
		1165	(incorrect result for the third desk-check)

Figure 6-16 Desk-check table for the incorrect algorithm in Figure 6-15

Second Logic Error: Reversing the Outer and Nested Decisions

Another common error made when writing selection structures is to reverse the decisions made by the outer and nested structures. Figure 6-17 shows an example of this error in the bonus algorithm. The correct algorithm is included in the figure for comparison. Unlike the selection structures in the correct algorithm, which determine the salesperson's code before determining his or her sales, the selection structures in the incorrect algorithm determine the sales before determining the code. Consider how this difference changes the algorithm. In the correct algorithm, the selection structures indicate that only salespeople who have a code of X receive an additional bonus. The selection structures in the incorrect algorithm, on the other hand, indicate that the additional bonus is given to all salespeople who have sales greater than or equal to 10000.

Correct algorithm

```
1. enter the code and sales
2. calculate the bonus by multiplying
   the sales by .08
3. if (the code is X)
       if (the sales are greater than or
       equal to 10000)
           add 150 to the bonus
       else
           add 125 to the bonus
       end if
   end if
4. display the bonus
```

Incorrect algorithm

```
1. enter the code and sales
2. calculate the bonus by multiplying
   the sales by .08
3. if (the sales are greater than or equal
   to 10000)
       if (the code is X)
           add 150 to the bonus
       else
           add 125 to the bonus
       end if
   end if
4. display the bonus
```

the outer and nested decisions are reversed

Figure 6-17 Correct algorithm and an incorrect algorithm containing the second logic error

Desk-checking the incorrect algorithm in Figure 6-17 will show you why the algorithm will not work correctly. The first instruction in the incorrect

algorithm is to enter the code and sales—in this case, X and 15000. The second instruction is to multiply the sales by .08; doing this results in a bonus of 1200. The condition in the outer selection structure in the third instruction determines whether the sales are greater than or equal to 10000. They are, so the nested selection structure's condition determines whether the salesperson's code is X. It is, so the nested selection structure's true path adds 150 to the bonus amount, giving 1350. After the bonus is calculated, both selection structures end. The last instruction in the incorrect algorithm displays the bonus (1350) on the screen. Even though its selection structures are phrased incorrectly, the incorrect algorithm produces the same result as the correct algorithm using the first set of test data.

Using the second set of test data, the user enters X as the code and 9000 as the sales. The second instruction in the incorrect algorithm multiplies the sales by .08; doing this results in a bonus of 720. The condition in the outer selection structure in the third instruction determines whether the sales are greater than or equal to 10000. They aren't, so the outer selection structure ends. Notice that the nested selection structure is not processed when the outer selection structure's condition evaluates to false. The last instruction in the incorrect algorithm displays 720 as the bonus, which is not correct.

Using the third set of test data, the user enters A as the code and 13000 as the sales. The second instruction in the incorrect algorithm calculates the bonus by multiplying the sales by .08; the answer is 1040. The condition in the outer selection structure in the third instruction determines whether the sales are greater than or equal to 10000. They are, so the nested selection structure's condition determines whether the salesperson's code is X. It isn't, so the nested selection structure's false path adds 125 to the bonus; this results in 1165. After the bonus is calculated, both selection structures end. The last instruction in the incorrect algorithm displays 1165 as the bonus, which is not correct. Figure 6-18 shows the desk-check table for the incorrect algorithm. As indicated in the figure, only the result of the first desk-check is correct.

code	sales	bonus	
~~X~~	~~15000~~	~~1200~~	
		~~1350~~	(correct result for the first desk-check)
~~X~~	~~9000~~	~~720~~	(incorrect result for the second desk-check)
A	13000	~~1040~~	
		1165	(incorrect result for the third desk-check)

Figure 6-18 Desk-check table for the incorrect algorithm in Figure 6-17

Third Logic Error: Using an Unnecessary Nested Selection Structure

Another error often made when writing selection structures is to include an unnecessary nested selection structure. In almost all cases, a selection structure containing this error still will produce the correct results. The only problem is that it does so less efficiently than selection structures that are properly structured. Figure 6-19 shows an example of this error in the bonus algorithm. The correct algorithm is included in the figure for comparison. Unlike the correct algorithm, the inefficient algorithm contains three (rather than two)

selection structures. Notice that the condition in the third selection structure determines whether the sales are less than 10000 and is processed only when the condition in the second selection structure is false. In other words, it is processed only when the sales are not greater than or equal to 10000. However, if the sales are not greater than or equal to 10000, then they would have to be less than 10000; so the third selection structure is unnecessary.

Correct algorithm

```
1. enter the code and sales
2. calculate the bonus by multiplying
   the sales by .08
3. if (the code is X)
       if (the sales are greater than or
       equal to 10000)
           add 150 to the bonus
       else
           add 125 to the bonus
       end if
   end if
4. display the bonus
```

Incorrect algorithm

```
1. enter the code and sales
2. calculate the bonus by multiplying
   the sales by .08
3. if (the code is X)
       if (the sales are greater than or
       equal to 10000)
           add 150 to the bonus
       else
           if (the sales are less than 10000)
               add 125 to the bonus
           end if
       end if
   end if
4. display the bonus
```

unnecessary nested selection structure

Figure 6-19 Correct algorithm and an incorrect algorithm containing the third logic error

Desk-checking the inefficient algorithm in Figure 6-19 will help you understand why the last selection structure is unnecessary. The first instruction in the inefficient algorithm is to enter the code and sales—in this case, X and 15000. The second instruction is to calculate the bonus by multiplying the sales by .08; the answer is 1200. The condition in the first selection structure determines whether the salesperson's code is X. It is, so the condition in the second selection structure checks whether the sales are greater than or equal to 10000. They are, so the second selection structure's true path adds 150 to the bonus amount, giving 1350. The last instruction in the inefficient algorithm displays 1350 as the bonus, which is correct.

Using the second set of test data, the user enters X as the code and 9000 as the sales. The second instruction in the inefficient algorithm calculates the bonus by multiplying the sales by .08, giving 720. The condition in the first selection structure determines whether the salesperson's code is X. It is, so the condition in the second selection structure checks whether the sales are greater than or equal to 10000. They aren't, so the condition in the third selection structure determines whether the sales are less than 10000—an unnecessary decision. In this case, the sales are less than 10000, so the third selection structure's true path adds 125 to the bonus, giving 845. The last instruction in the inefficient algorithm displays 845 as the bonus, which is correct.

Using the third set of test data, the user enters A as the code and 13000 as the sales. The second instruction in the inefficient algorithm multiplies the sales by .08, resulting in 1040 as the bonus. The condition in the first selection structure determines whether the salesperson's code is X. It's not, so the first selection structure ends. Notice that the two nested selection structures are

not processed when the first selection structure's condition evaluates to false. The last instruction in the inefficient algorithm displays 1040 as the bonus, which is correct. Figure 6-20 shows the desk-check table for the inefficient algorithm. Although the results of the three desk-checks are correct, the result of the second desk-check is obtained in a less efficient manner.

code	sales	bonus	
~~X~~	~~15000~~	~~1200~~	
		~~1350~~	(correct result for the first desk-check)
~~X~~	~~9000~~	~~720~~	
		~~845~~	(correct but inefficient result for the second desk-check)
A	13000	1040	(correct result for the third desk-check)

Figure 6-20 Desk-check table for the inefficient algorithm in Figure 6-19



The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 6-2

1. List the three errors commonly made when writing selection structures.

2. Which of the errors from Question 1 makes the selection structure inefficient but still produces the correct results?

Multiple-Alternative Selection Structures

Figure 6-21 shows the problem specification and IPO chart for the Kindlon High School problem. The problem's solution requires a selection structure that can choose from several alternatives; in this case, it can choose from several different letter grades. As the figure indicates, when the letter grade is A, the selection structure should display the message "Excellent". When the letter grade is B, the selection structure should display the message "Above Average", and so on. Selection structures containing several alternatives are referred to as **multiple-alternative selection structures** or **extended selection structures**.

Problem specification
Mr. Jacoby teaches math at Kindlon High School. He wants a program that displays a message based on a letter grade he enters. The valid letter grades and their corresponding messages are shown below. If the letter grade is not valid, the program should display the "Invalid grade" message.

Letter grade	Message
A	Excellent
B	Above Average
C	Average
D	Below Average
F	Below Average

Figure 6-21 Problem specification and IPO chart for the Kindlon High School problem *(continues)*

(continued)

Input	Processing	Output
grade	Processing items: none	message

```
Algorithm:
1. enter the grade
2. if (the grade is one of the following:)
        A        display "Excellent" message
        B        display "Above Average" message
        C        display "Average" message
        D or F   display "Below Average" message
   else
        display "Invalid grade"
   end if
```

Figure 6-21 Problem specification and IPO chart for the Kindlon High School problem

Figure 6-22 shows the Kindlon High School algorithm in flowchart form. The diamond in the flowchart represents the multiple-alternative selection structure's condition. Recall that the diamond also is used to represent the condition in both single-alternative and dual-alternative selection structures. However, unlike the diamond in both of those selection structures, the diamond in a multiple-alternative selection structure has several flowlines (rather than only two flowlines) leading out of the symbol. Each flowline represents a possible path and must be marked appropriately, indicating the value or values necessary for the path to be chosen.

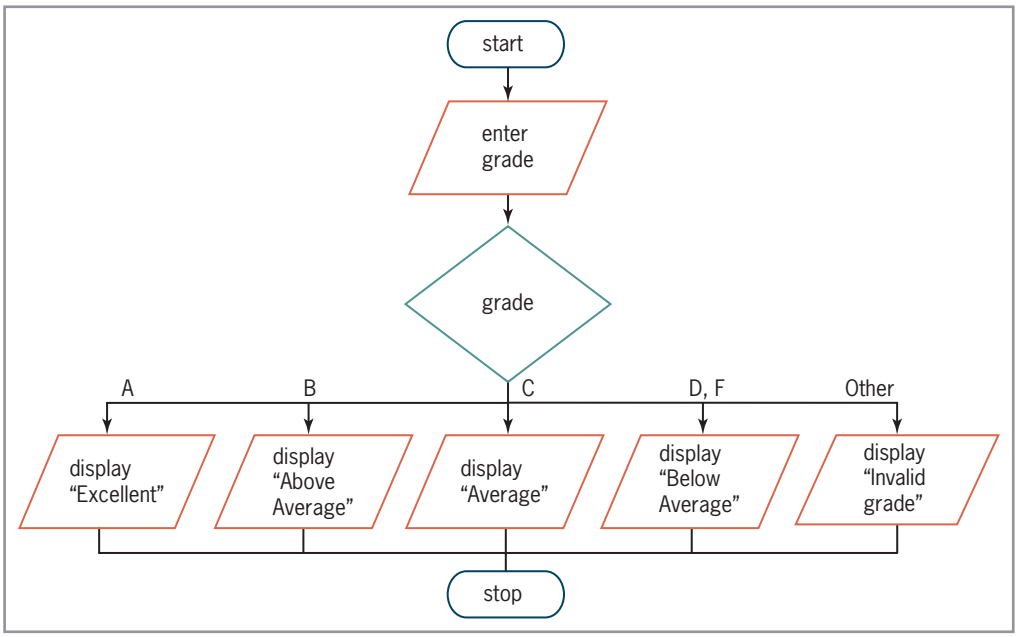


Figure 6-22 Flowchart for the Kindlon High School problem

Figure 6-23 shows two versions of the C++ code for the Kindlon High School program. Both versions use the multiple-alternative form of the `if` statement to code the multiple-alternative selection structure. The multiple-alternative form contains several `if` and `else` clauses. Although both versions produce the same result, the second version provides a more convenient way of coding a multiple-alternative selection structure.

Version 1

```
int main()
{
    char grade = ' ';

    //enter grade
    cout << "Letter grade: ";
    cin >> grade;
    grade = toupper(grade);

    if (grade == 'A')
        cout << "Excellent";
    else
        if (grade == 'B')
            cout << "Above Average";
        else
            if (grade == 'C')
                cout << "Average";
            else
                if (grade == 'D' || grade == 'F')
                    cout << "Below Average";
                else     //default
                    cout << "Invalid grade";
                //end if
            //end if
        //end if
    //end if

    cout << endl;
    system("pause");
    return 0;
}   //end of main function
```

you get here when the grade is not A

you get here when the grade is not A and not B

you get here when the grade is not A, B, or C

you get here when the grade is not A, B, C, D, or F

Version 2

```
int main()
{
    char grade = ' ';

    //enter grade
    cout << "Letter grade: ";
    cin >> grade;
    grade = toupper(grade);

    if (grade == 'A')
        cout << "Excellent";
    else if (grade == 'B')
        cout << "Above Average";
    else if (grade == 'C')
        cout << "Average";
    else if (grade == 'D' || grade == 'F')
        cout << "Below Average";
    else     //default
        cout << "Invalid grade";
    //end if

    cout << endl;
    system("pause");
    return 0;
}   //end of main function
```

you can use one comment to mark the end of the entire structure

Figure 6-23 Two versions of the C++ code for the Kindlon High School problem

The `switch` Statement

Instead of using the multiple-alternative form of the `if` statement to code a multiple-alternative selection structure in C++, you sometimes (but not always) can use the **switch statement**. Figure 6-24 shows the `switch` statement's syntax and includes an example of using the statement (rather than the `if` statement) in the Kindlon High School code from Figure 6-23. As the syntax shows, the `switch` statement begins with the `switch` clause, which contains a *selectorExpression* enclosed in parentheses. The *selectorExpression* can contain any combination of variables, constants, functions, and operators; however, the combination must result in a value whose data type is `bool`, `char`, `short`, `int`, or `long`. In the example in Figure 6-24, the *selectorExpression* is a `char` variable named `grade`. Following the `switch` clause is a statement block. Recall from Chapter 5 that a statement block is one or more statements enclosed in a set of braces. Between the `switch` statement's opening and closing braces are one or more `case` clauses; you can include as many `case` clauses as necessary. Each `case` clause in a `switch` statement represents a different alternative. Notice that each `case` clause contains a value followed by a colon. The value can be a literal constant, a named constant, or an expression composed of literal and named constants. The data type of the value should be compatible with the data type of the *selectorExpression*. When the *selectorExpression* is numeric, the values in the `case` clauses should be numeric. Likewise, when the *selectorExpression* is a character, the values should be characters. In the example in Figure 6-24, the data type of the *selectorExpression* is `char`, and so is the data type of the values in the `case` clauses (`'A'`, `'B'`, `'C'`, `'D'`, and `'F'`). Following the colon in each `case` clause are one or more statements that are processed when the *selectorExpression* matches that `case`'s value. After the computer processes the instructions in a `case` clause, you typically want the computer to leave the `switch` statement without processing the remaining instructions in the statement. You do this by including the `break` statement as the last statement in the `case` clause. The **break statement** tells the computer to leave ("break out of") the `switch` statement at that point. If you do not use the `break` statement in a `case` clause, the computer continues processing the remaining instructions in the `switch` statement; this may or may not be what you intended. After processing the `break` statement, the computer processes the instruction that follows the `switch` statement's closing brace. For clarity, it is a good programming practice to document the end of the `switch` statement with a comment, such as `//end switch`. In addition to the `case` clauses, you also can include one `default` clause in a `switch` statement. Although the `default` clause can appear anywhere within the `switch` statement, it usually is entered as the last clause in the statement. When it is in that position, it does not need a `break` statement; however, some programmers include the `break` statement for clarity. If the `default` clause is not the last clause, a `break` statement is required in order to stop the computer from processing the instructions in the next `case` clause.



Recall that `char` literal constants are enclosed in single quotation marks.



In Computer Exercises 19 and 20, you will observe the result of not using the `break` statement to break out of the `switch` statement.

HOW TO Use the `switch` Statement

Syntax

switch (*selectorExpression*)
{
case *value1*:
    one or more statements
    [**break;**]
[**case** *value2*:
    one or more statements
    [**break;**]]
[**case** *valueN*:
    one or more statements
    [**break;**]]
[**default:**
    one or more statements to be processed when the selector-Expression does not match any of the values in the `case` *clauses*
    [**break;**]]
} `//end switch`

Example

```
int main()
{
    char grade = ' ';

    //enter grade
    cout << "Letter grade: ";
    cin >> grade;
    grade = toupper(grade);

    switch (grade)
    {
    case 'A':
        cout << "Excellent";
        break;
    case 'B':
        cout << "Above Average";
        break;
    case 'C':
        cout << "Average";
        break;
    case 'D':
    case 'F':
        cout << "Below Average";
        break;
    default:
        cout << "Invalid grade";
    }   //end switch

    cout << endl;
    system("pause");
    return 0;
}   //end of main function
```

Figure 6-24 How to use the `switch` statement

Desk-checking the code in Figure 6-24 will help you understand how the `switch` statement is processed by the computer. You will desk-check the code using the following three letter grades: b, D, and x. First, the code declares a `char` variable named `grade`. It then prompts the user to enter a letter grade and assigns the user's response—in this case, the letter b—to the `grade` variable. Next, the `grade = toupper(grade);` statement assigns the letter B to the `grade` variable. The `switch` statement is processed next; notice that the `grade` variable is used as the statement's *selectorExpression*. When processing the `switch` statement, the computer compares the value of the *selectorExpression* with the value listed in each of the `case` clauses, one `case` clause at a time beginning with the first. If a match is found, the computer processes the instructions contained in that `case` clause, stopping only when it encounters either a `break` statement or the `switch` statement's closing brace; the computer then skips to the instruction following the closing brace. In this case, the computer first compares the value of the *selectorExpression* (`'B'`) with the value listed in the first `case` clause (`'A'`). `'B'` does not match `'A'`, so the computer compares the *selectorExpression*'s value (`'B'`) with the value listed in the second `case` clause (`'B'`). In this case, `'B'` matches `'B'`, so the computer processes the instructions contained in the second `case` clause. The first statement in the second `case` clause displays the message "Above Average" on the screen. The next statement, `break;`, tells the computer to skip the remaining instructions in the `switch` statement and continue processing with the instruction that follows the `switch` statement's closing brace.

Now use the letter D to desk-check the code. First, the code declares a `char` variable named `grade`. It then prompts the user to enter a letter grade and assigns the user's response—in this case, the letter D—to the variable. The `grade = toupper(grade);` statement also assigns the letter D to the variable. The `switch` statement is processed next. When processing the `switch` statement, the computer compares the value of the *selectorExpression* (`'D'`) with the value listed in the first `case` clause (`'A'`). `'D'` does not match `'A'`, so the computer compares the *selectorExpression*'s value (`'D'`) with the value listed in the second `case` clause (`'B'`). `'D'` does not match `'B'`, so the computer compares the *selectorExpression*'s value (`'D'`) with the value listed in the third `case` clause (`'C'`). `'D'` does not match `'C'`, so the computer compares the *selectorExpression*'s value (`'D'`) with the value listed in the fourth `case` clause (`'D'`); here, the computer finds a match. However, notice that there is no statement immediately below the `case 'D':` clause. So, what (if anything) will appear when the grade is D? Recall that, when the value of the *selectorExpression* matches the value in a `case` clause, the computer processes the instructions contained in that clause until it encounters either a `break` statement or the `switch` statement's closing brace. In this case, not finding any instructions in the `case 'D':` clause, the computer continues processing with the instructions in the next clause, which is the `case 'F':` clause. The first instruction in the `case 'F':` clause displays the message "Below Average", which is the correct message to display. The second instruction tells the computer to break out of the `switch` statement. In other words, the `cout << "Below Average";` and `break;` statements are processed when the grade is either D or F. The computer then skips to the instruction following the `switch` statement's closing brace. As this example shows, you can process the same instructions for more than one value by listing each value in a separate `case` clause, as long as the clauses appear together in

the `switch` statement. The last `case` clause in the group of related clauses should contain the instructions you want the computer to process when one of the values in the group matches the *selectorExpression*. Only the last `case` clause in the group of related clauses should contain the `break` statement.

Finally, you will use the letter x to desk-check the code. After declaring the `grade` variable, the code prompts the user to enter a letter grade, and it assigns the user's response—in this case, the letter x—to the variable. Next, the `grade = toupper(grade);` statement assigns the letter X to the variable. The `switch` statement is processed next. When processing the `switch` statement, the computer compares the value of the *selectorExpression* (`'X'`) with the value listed in each of the `case` clauses, beginning with the first `case` clause. Notice that the letter X does not appear as a value in any of the `case` clauses. When the *selectorExpression* does not match any of the values listed in the `case` clauses, the computer processes the instructions contained in the `default` clause. If there is no `default` clause, the computer skips to the instruction following the `switch` statement's closing brace. In the example in Figure 6-24, the instruction in the `default` clause displays the message "Invalid grade" on the screen. If the `default` clause is the last clause in the `switch` statement, as it is in Figure 6-24, the computer then skips to the instruction following the `switch` statement's closing brace.

Figure 6-25 shows another problem description whose solution requires a multiple-alternative selection structure. (The flowchart for the program is contained in the Ch6Flowcharts.pdf file, which is located in the Cpp\Chap06 folder.) In this case, the multiple-alternative selection structure uses a product's ID to determine its price. The figure also includes the problem's IPO chart information and C++ instructions. Notice that the multiple-alternative selection structure is coded using the `switch` statement, which is shaded in the figure. The `switch` statement assigns 50.55 to the `price` variable when the product ID, which is stored in the `productId` variable, is 1. When the product ID is either 2 or 9, the `switch` statement assigns 12.35 to the `price` variable. It assigns 11.46 to the `price` variable when the product ID is one of the following: 5, 7, or 11. Finally, it assigns -1 to the `price` variable when the product ID is anything other than 1, 2, 5, 7, 9, or 11. The `if` statement in the code compares the value stored in the `price` variable with the number -1 and then displays either the "Invalid product ID" message or the product's price.

Problem specification
The sales manager at Warren Company wants a program that displays a price based on a product ID she enters. The valid product IDs and their corresponding prices are shown here. If the product ID is not valid, the program should display the "Invalid product ID" message.

Product ID	Price
1	50.55
2	12.35
5	11.46
7	11.46
9	12.35
11	11.46

Figure 6-25 Problem specification, IPO chart information, and C++ instructions for the Warren Company problem (*continues*)

(continued)

IPO chart information	C++ instructions
Input	
product ID	`int productId = 0;`
Processing	
none	
Output	
price	`double price = 0.0;`
Algorithm	
1. enter the product ID	`cout << "Product ID (1, 2, 5, 7, 9, or 11): ";` `cin >> productId;`

2. if (the product ID is one of the following:)
1 assign 50.55 as the price
2 or 9 assign 12.35 as the price
5, 7, or 11 assign 11.46 as the price
else
assign -1 as the price
end if
	`switch (productId)`
	`{`
	`case 1:`
	`price = 50.55;`
	`break;`
	`case 2:`
	`case 9:`
	`price = 12.35;`
	`break;`
	`case 5:`
	`case 7:`
	`case 11:`
	`price = 11.46;`
	`break;`
	`default:`
	`price = -1;`
	`} //end switch`

3. if (the price is -1)
display "Invalid product ID" message
else
display the price
end if
	`if (price == -1)`
	`cout << "Invalid product ID" << endl;`
	`else`
	`cout << "Price: $" << price << endl;`
	`//end if`

Figure 6-25 Problem specification, IPO chart information, and C++ instructions for the Warren Company problem

Mini-Quiz 6-3



The answers to Mini-Quiz questions are located in Appendix A.

1. A selection structure should display the message "Great score!" when a student's test score is at least 90. When the test score is 70 through 89, the selection structure should display the message "Good score". When the test score is 0 through 69, the selection structure should display the message "Retake the test". For all other test scores, the selection structure should display the message "Invalid test score". Write the appropriate C++ code using the more convenient form of the `if` statement.

2. If a `switch` statement's *selectorExpression* is an `int` variable named `stateCode`, which of the following `case` clauses will be processed when the `stateCode` variable contains the number 2?

   a. `case "2":`

   b. `case 2:`

   c. `case = 2;`

   d. `case == 2;`

3. The _____ statement tells the computer to leave the `switch` statement at that point.



The answers to the labs are located in Appendix A.



LAB 6-1 Stop and Analyze

Study the flowchart shown in Figure 6-26, and then answer the questions.

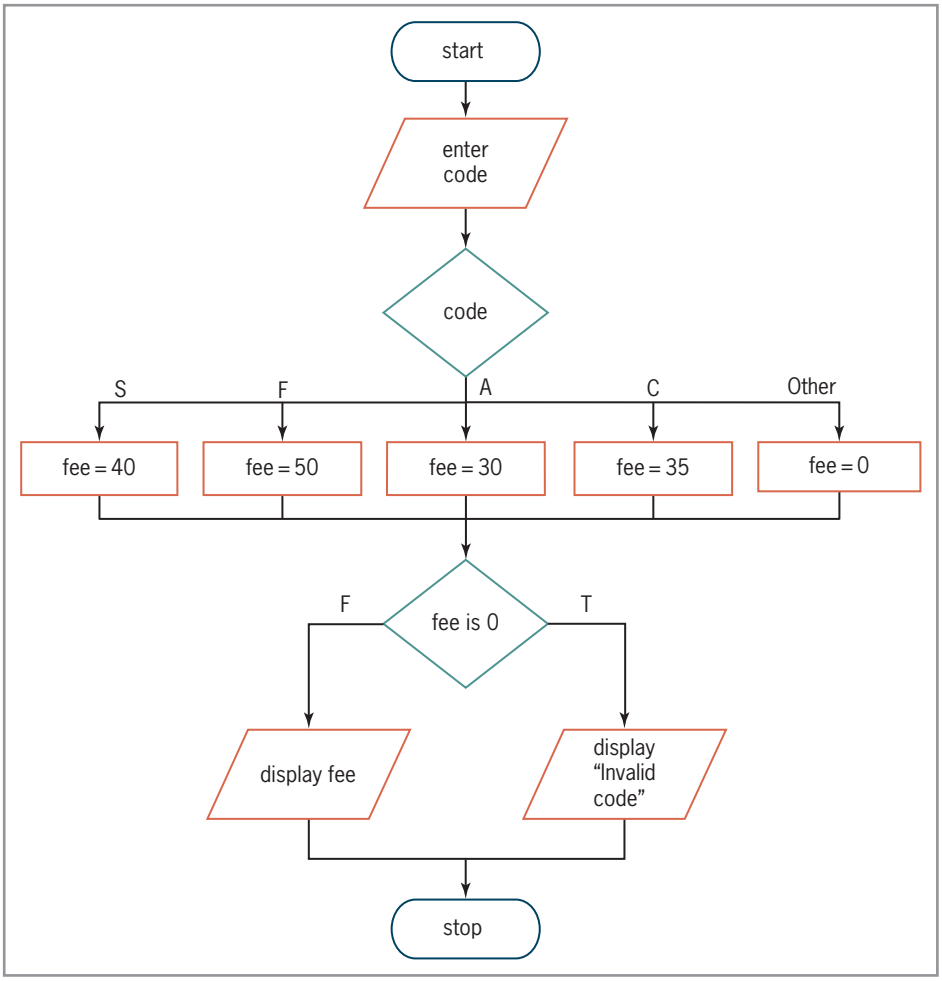


Figure 6-26 Flowchart for Lab 6-1

QUESTIONS

1. What will the program display when the code is C?
2. How can you write the multiple-alternative selection structure using the longer form of the `if` statement?
3. How can you write the multiple-alternative selection structure using the shorter form of the `if` statement?
4. How can you write the multiple-alternative selection structure using the `switch` statement?
5. What changes would you need to make to the code from Question 4 so that each `case` clause displays the appropriate fee, and the `default` clause displays the "Invalid code" message?



LAB 6-2 Plan and Create

In this lab, you will plan and create an algorithm for Golf Pro. The problem specification and examples of calculations are shown in Figure 6-27.

Problem specification
Jennifer Yardley is the owner of Golf Pro, a U.S. company that sells golf equipment both domestically and abroad. She wants a program that displays the amount of a salesperson's commission. A commission is a percentage of the sales made by the salesperson. Some companies use a fixed rate to calculate the commission, while others (like Golf Pro) use a rate that varies with the amount of sales. Golf Pro's commission schedule is shown below, along with examples of using the schedule to calculate the commission on three different sales amounts. Notice that the commission for each range in the schedule is calculated differently. The commission for sales in the first range is calculated by multiplying the sales by 2%. As Example 1 shows, if the sales are $15000, the commission is $300. The commission for sales in the second range is calculated by multiplying the sales over $100,000 by 5% and then adding $2000 to the result. As Example 2 shows, if the sales are $250,000, the commission is $9500. The commission for sales starting at $400,001 is calculated by multiplying the sales over $400,000 by 10% and then adding $17000 to the result. Example 3 indicates that the commission for sales of $500,000 is $27000. If the sales are less than zero, the program should display the message "The sales cannot be less than 0."

Figure 6-27 Problem specification and calculation examples for Lab 6-2 (*continues*)

(continued)

Sales range	Commission
$0 - 100,000	multiply the sales by 2%
$100,001 – 400,000	multiply the sales over 100,000 by 5% and then add 2000 to the result
$400,001 and over	multiply the sales over 400,000 by 10% and then add 17000 to the result

Example 1
Sales: $15000
Commission: 15000 * .02 = 300

Example 2
Sales: $250,000
Commission: (250,000 - 100,000) * .05 + 2000 = 9500

Example 3
Sales: $500,000
Commission: (500,000 - 400,000) * .1 + 17000 = 27000

Figure 6-27 Problem specification and calculation examples for Lab 6-2

First, analyze the problem, looking for the output first and then for the input. In this case, the user wants the program to display a salesperson's commission. To calculate the commission, the computer will need to know the salesperson's sales and how to calculate the commission. The sales amount will be entered by the user. The calculation instructions, on the other hand, are specified in the problem specification. Next, plan the algorithm. Recall that most algorithms begin with an instruction to enter the input items into the computer, followed by instructions that process the input items, typically including the items in one or more calculations. Most algorithms end with one or more instructions that display, print, or store the output items. Figure 6-28 shows the completed IPO chart for the Golf Pro problem.

Input	Processing	Output
sales	Processing items: none	commission

```
Algorithm:
1. enter the sales
2. if (the sales are less than 0)
       commission = -1
   else if (the sales are one of the following:)
       0 through 100,000
              commission = sales * .02
       100,001 through 400,000
              commission = (sales - 100,000) * .05 + 2000
       400,001 and over
              commission = (sales - 400,000) * .1 + 17000
   end if
3. if (the commission is not -1)
       display the commission
   else
       display an error message
   end if
```

Figure 6-28 Completed IPO chart for the Golf Pro problem

After completing the IPO chart, you then move on to the third step in the problem-solving process, which is to desk-check the algorithm. You will desk-check the Golf Pro algorithm four times, using sales of 15000, 250,000, 500,000, and -500. Figure 6-29 shows the completed desk-check table. Notice that the amounts in the commission column for the first three desk-checks agree with the results of the manual calculations from Figure 6-27.

sales	commission
~~15000~~	~~300~~
~~250000~~	~~9500~~
~~500000~~	~~27000~~
-500	-1

Figure 6-29 Completed desk-check table for the Golf Pro algorithm

The fourth step in the problem-solving process is to code the algorithm into a program. You begin by declaring memory locations that will store the values of the input, processing (if any), and output items. The Golf Pro problem will need two memory locations to store the input and output items. The input item (sales) will be stored in a variable, because the user should be allowed to change its value during runtime. The output item (commission) also will be stored in a variable, because its value is based on the current value of the input item. The sales amount will always be an integer, so you will store it in an `int` variable. You will store the commission in a `double` variable, because its value will be a real number. Figure 6-30 shows the IPO chart information and corresponding C++ instructions.

IPO chart information	**C++ instructions**
Input	
sales	`int sales = 0;`
Processing	
none	
Output	
commission	`double commission = 0.0;`

Figure 6-30 IPO chart information and C++ instructions for the Golf Pro problem (*continues*)

(continued)

Algorithm

```
1. enter the sales                          cout << "Sales: ";
                                            cin >> sales;

2. if (the sales are less than 0)           if (sales < 0)
     commission = -1                          commission = -1;
   else if (the sales are one of the        else if (sales <= 100000)
     following:)                              commission = sales * .02;
     0 through 100,000                      else if (sales <= 400000)
       commission = sales * .02               commission = (sales - 100000)
     100,001 through 400,000                  * .05 + 2000;
       commission = (sales - 100,000)       else
       * .05 + 2000                           commission = (sales - 400000)
     400,001 and over                         * .1 + 17000;
       commission = (sales - 400,000)       //end if
       * .1 + 17000
   end if

3. if (the commission is not -1)            if (commission != -1)
     display the commission                   cout << "Commission: $" <<
   else                                       commission << endl;
     display an error message               else
   end if                                     cout << "The sales cannot
                                              be less than 0." << endl;
                                            //end if
```

Figure 6-30 IPO chart information and C++ instructions for the Golf Pro problem

The fifth step in the problem-solving process is to desk-check the program. You begin by placing the names of the declared variables and named constants (if any) in a new desk-check table, along with their initial values. You then desk-check the remaining C++ instructions in order, recording in the desk-check table any changes made to the variables. Figure 6-31 shows the completed desk-check table for the Golf Pro program. The results agree with those shown in the algorithm's desk-check table in Figure 6-29.

	sales	commission
first desk-check	~~0~~	~~0.0~~
	~~15000~~	~~300.0~~
second desk-check	~~0~~	~~0.0~~
	~~250000~~	~~9500.0~~
third desk-check	~~0~~	~~0.0~~
	~~500000~~	~~27000.0~~
fourth desk-check	~~0~~	~~0.0~~
	-500	-1.0

Figure 6-31 Completed desk-check table for the Golf Pro program

The final step in the problem-solving process is to evaluate and modify (if necessary) the program. Recall that you evaluate a program by entering its instructions into the computer, and then using the computer to run (execute) it. While the program is running, you enter the same sample data used when desk-checking the program.

DIRECTIONS

Follow the instructions for starting your C++ development tool. Depending on the development tool you are using, you may need to create a new project; if so, name the project Lab6-2 Project and save it in the Cpp6\Chap06 folder. Enter the instructions shown in Figure 6-32 in a source file named Lab6-2.cpp. (Do not enter the line numbers.) Save the file in either the project folder or the Cpp6\Chap06 folder. Now follow the appropriate instructions for running the Lab6-2.cpp file. Use the sample data from Figure 6-31 to test the program. If necessary, correct any bugs (errors) in the program.

```
 1  //Lab6-2.cpp - displays a salesperson's commission
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <iomanip>
 6  using namespace std;
 7
 8  int main()
 9  {
10     //declare variables
11     int sales          = 0;
12     double commission = 0.0;
13
14     //enter input
15     cout << "Sales: ";
16     cin >> sales;
17
18     //determine commission
19     if (sales < 0)
20        commission = -1;
21     else if (sales <= 100000)
22        commission = sales * .02;
23     else if (sales <= 400000)
24        commission = (sales - 100000) * .05 + 2000;
25     else
26        commission = (sales - 400000) * .1 + 17000;
27     //end if
28
29     //display commission or error message
30     if (commission != -1)
31     {
32        cout << fixed << setprecision(2);
33        cout << "Commission: $" << commission << endl;
34     }
35     else
36        cout << "The sales cannot be less than 0." << endl;
37     //end if
38
39     system("pause");
40     return 0;
41  }  //end of main function
```

if your C++ development tool does not require this statement, either omit it or make it a comment

Figure 6-32 Golf Pro program



LAB 6-3 Modify

If necessary, create a new project named Lab6-3 Project. Enter (or copy) the Lab6-2.cpp instructions into a new source file named Lab6-3.cpp. Change Lab6-2.cpp in the first comment to Lab6-3.cpp. Golf Pro now uses the information shown in Figure 6-33 to determine a salesperson's commission. Modify the program appropriately. Use the `switch` statement to determine the commission. If the sales are less than zero, display the message "The sales cannot be less than 0." If the code is not 1, 2, or 3, display the message "Invalid code" and don't ask the user for a sales amount. Test the program three times using the test data included in Figure 6-33. Now test it a fourth time using an invalid code. Finally, test it a fifth time using an invalid sales amount.

Code	Commission
1	multiply the sales by 2%
2	multiply the sales over 100,000 by 5% and then add 2000 to the result
3	multiply the sales over 400,000 by 10% and then add 17000 to the result

Example 1
Code: 1
Sales: $1000
Commission: 1000 * .02 = 20

Example 2
Code: 2
Sales: $110,000
Commission: (110,000 - 100,000) * .05 + 2000 = 2500

Example 3
Code: 3
Sales: $500,000
Commission: (500,000 - 400,000) * .1 + 17000 = 27000

Figure 6-33 Problem specification and calculation examples for Lab 6-3



LAB 6-4 Desk-Check

Desk-check the code shown in Figure 6-34 three times, using the numbers 2, 5, and 100.

```
//declare variable
int number = 0;

//enter input item
cout << "Enter a number: ";
cin >> number;

//perform calculations
switch (number)
{
      case 1:
      case 2:
      case 3:
            number = number * 2;
            break;
      case 4:
      case 5:
            number = number + 5;
            break;
      default:
            number = number - 50;
}     //end switch

//display number
cout << "Final number: " << number << endl;
```

Figure 6-34 Code for Lab 6-4



LAB 6-5 Debug

Follow the instructions for starting C++ and running the program contained in the Lab6-5.cpp file. The file is contained in either the Cpp6\Chap06\Lab6-5 Project folder or the Cpp6\Chap06 folder. Test the program using the following codes: 1, 2, 3, 4, 5, 9, and -3. Debug the program.

Summary

- You can nest a selection structure within either the true or false path of another selection structure.
- Logic errors commonly made when writing selection structures usually are a result of one of the following three mistakes: using a compound condition rather than a nested selection structure, reversing the decisions in the outer and nested selection structures, or using an unnecessary nested selection structure.
- Some solutions require selection structures that can choose from several alternatives. The selection structures are commonly referred to as multiple-alternative selection structures or extended selection structures. You can code these selection structures using either the multiple-alternative form of the `if` statement or the `switch` statement.
- In a flowchart, a diamond is used to represent the condition in a multiple-alternative selection structure. The diamond has one flowline leading into the symbol and several flowlines leading out of the symbol. Each flowline leading out of the diamond represents a possible path and must be marked to indicate the value or values necessary for the path to be chosen.
- In a `switch` statement, the data type of the value in each `case` clause should be compatible with the data type of the statement's *selectorExpression*. The *selectorExpression* must evaluate to a value whose data type is `bool`, `char`, `short`, `int`, or `long`.
- Most `case` clauses in a `switch` statement contain a `break` statement, which tells the computer to leave the `switch` statement.
- It is a good programming practice to include a comment (such as `//end switch`) to identify the end of a `switch` statement in a program.

Key Terms

`break` statement—a C++ statement used to tell the computer to leave a `switch` statement

Extended selection structures—another name for multiple-alternative selection structures

Multiple-alternative selection structures—selection structures that contain several alternatives; also called extended selection structures; can be coded using either the multiple-alternative form of the `if` statement or the `switch` statement

Nested selection structure—a selection structure that is wholly contained (nested) within another selection structure

`switch` statement—a C++ statement that can be used to code a multiple-alternative selection structure

Review Questions

Use the code shown in Figure 6-35 to answer Review Questions 1 through 3.

```
int number = 0;
cout << "Number: ";
cin >> number;

if (number <= 100)
     number = number * 2;
else
     if (number > 500)
          number = number * 3;
     //end if
//end if
```

Figure 6-35

1. If the `number` variable contains the integer 90, what value will be in the `number` variable after the code shown in Figure 6-35 is processed?

   a. 0

   b. 90

   c. 180

   d. 270

2. If the `number` variable contains the integer 1000, what value will be in the `number` variable after the code shown in Figure 6-35 is processed?

   a. 0

   b. 1000

   c. 2000

   d. 3000

3. If the `number` variable contains the integer 200, what value will be in the `number` variable after the code shown in Figure 6-35 is processed?

   a. 0

   b. 200

   c. 400

   d. 600

Use the code shown in Figure 6-36 to answer Review Questions 4 through 7.

```
int id = 0;
cout << "ID: ";
cin >> id;

if (id == 1)
     cout << "Janet";
else if (id == 2 || id == 3)
     cout << "Mark";
else if (id == 4)
     cout << "Jerry";
else
     cout << "Sue";
//end if
```

Figure 6-36

4. What will the code in Figure 6-36 display when the `id` variable contains the number 2?

   a. Janet

   b. Jerry

   c. Mark

   d. Sue

5. What will the code in Figure 6-36 display when the `id` variable contains the number 4?

   a. Janet

   b. Jerry

   c. Mark

   d. Sue

6. What will the code in Figure 6-36 display when the `id` variable contains the number 3?

   a. Janet

   b. Jerry

   c. Mark

   d. Sue

7. What will the code in Figure 6-36 display when the `id` variable contains the number 8?

   a. Janet

   b. Jerry

   c. Mark

   d. Sue

Use the code shown in Figure 6-37 to answer Review Questions 8 through 10.

```
int id = 0;
cout << "ID: ";
cin >> id;

switch (id)
{
     case 1:
          cout << "Janet";
          break;
     case 2:
          cout << "Mark";
          break;
     case 3:
     case 5:
          cout << "Jerry";
          break;
     default:
          cout << "Sue";
}    //end switch
```

Figure 6-37

8. What will the code in Figure 6-37 display when the `id` variable contains the number 2?

   a. Janet

   b. Jerry

   c. Mark

   d. Sue

9. What will the code in Figure 6-37 display when the `id` variable contains the number 4?

   a. Janet

   b. Jerry

   c. Mark

   d. Sue

10. What will the code in Figure 6-37 display when the `id` variable contains the number 3?

    a. Janet

    b. Jerry

    c. Mark

    d. Sue

Exercises



Pencil and Paper

TRY THIS

1. Using the multiple-alternative form of the `if` statement, write the C++ code for a multiple-alternative selection structure that displays the month corresponding to the number entered by the user. The number is stored in an `int` variable named `monthNum`. For example, if the `monthNum` variable contains the number 1, the selection structure should display the string "January". If the month number is not 1 through 12, display the "Incorrect month number" message. (The answers to TRY THIS Exercises are located at the end of the chapter.)

TRY THIS

2. Using the `switch` statement, write the C++ code that corresponds to the partial flowchart shown in Figure 6-38. Use a `char` variable named `code` and a double variable named `rate`. (The answers to TRY THIS Exercises are located at the end of the chapter.)

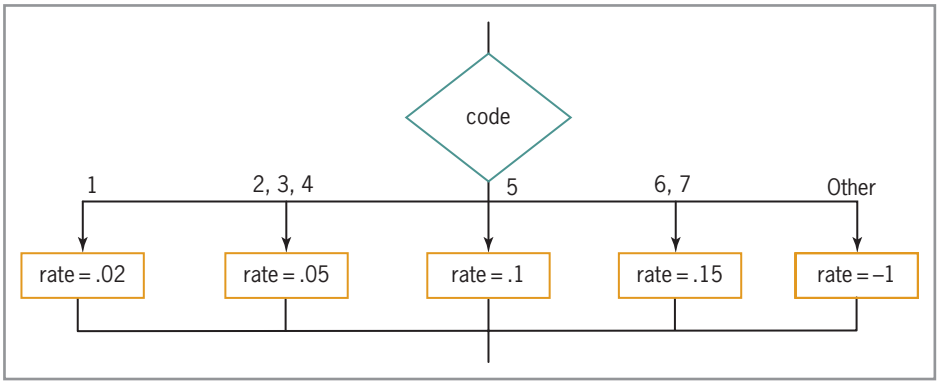


Figure 6-38

MODIFY THIS

3. Complete TRY THIS Exercise 1, and then rewrite the code using the `switch` statement.

MODIFY THIS

4. Complete TRY THIS Exercise 2, and then change the `switch` statement to the multiple-alternative form of the `if` statement.

INTRODUCTORY

5. Write the C++ code to display the message "Entry error" when the value in the `units` variable is less than or equal to 0. Otherwise, calculate the total owed as follows: If the `units` variable's value is less than 10, multiply the value by $5; otherwise, multiply it by $10. Store the total owed in the `total` variable.

INTRODUCTORY

6. A program stores sales amounts in two `int` variables named `marySales` and `jimSales`. Write the C++ code to display the message "Mary and Jim sold the same amount" when both variables contain the same number. If both variables contain different numbers, the code should compare both numbers and then display either the message "Mary sold more than Jim" or the message "Jim sold more than Mary".

7. A program uses a `char` variable named `department` and two `double` variables named `salary` and `raise`. The `department` variable contains one of the following letters (entered in either uppercase or lowercase): A, B, C, or D. Employees in departments A and B are receiving a 2% raise. Employees in department C are receiving a 1.5% raise, and employees in department D are receiving a 3% raise. Using the `switch` statement, write the C++ code to calculate the appropriate raise amount.

INTERMEDIATE

8. A program uses a `char` variable named `membership` and an `int` variable named `age`. The `membership` variable contains one of the following letters (entered in either uppercase or lowercase): M or N. The letter M stands for *member*, and the letter N stands for *non-member*. The program should display the appropriate seminar fee, which is based on a person's membership status and age. The fee schedule is shown in Figure 6-39. Write the C++ code to display the fee.

ADVANCED

Seminar fee	Criteria
$10	Club member less than 65 years old
$5	Club member at least 65 years old
$20	Non-member

Figure 6-39

9. The C++ code shown in Figure 6-40 should display "Illinois" when the state code is 1, "Kentucky" when it's 2, "New Hampshire" when it's 3, "Vermont" when it's 4, and "Massachusetts" when it's 5. If the state code is not 1 through 5, the code should display the "Unknown state" message. Correct the errors in the code.

SWAT THE BUGS

```
char stateCode = ' ';
cout << "State code (1, 2, 3, 4, or 5): ";
cin >> stateCode;
switch (stateCode)
{
     case "1":
                 cout << "Illinois";
                 break;
     case '2':
                 break;
                 cout << "Kentucky";
     case 3:
                 cout << "New Hampshire";
     case 4:
                 cout << "Vermont";
     case 5:
                 cout << "Massachusetts";
                 break;
     default:
                 cout << "Unknown state";
}    //end switch
```

Figure 6-40



Computer

TRY THIS

10. Complete Figure 6-41 by writing the algorithm and corresponding C++ instructions. Employees with a pay code of either 4 or 9 receive a 5% raise. Employees with a pay code of either 2 or 10 receive a 4% raise. All other employees receive a 3% raise. Use the multiple-alternative form of the `if` statement to code the multiple-alternative selection structure. Enter the C++ instructions into a source file named TryThis10.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Display the new pay in fixed-point notation with two decimal places. Save and run the program. Test the program using 1 and 500 as the pay code and current pay. The new pay should be $515.00. Now test the program using the following four sets of input values: 4 and 450, 9 and 500, 2 and 625, 10 and 500. (The answers to TRY THIS Exercises are located at the end of the chapter.)

IPO chart information	**C++ instructions**
Input	
pay code	`int code = 0;`
current pay	`double currentPay = 0.0;`
raise rate 1 (3%)	`const double RATE1 = .03;`
raise rate 2 (4%)	`const double RATE2 = .04;`
raise rate 3 (5%)	`const double RATE3 = .05;`
Processing	
raise	`double raise = 0.0;`
Output	
new pay	`double newPay = 0.0;`
Algorithm	

Figure 6-41

TRY THIS

11. Code the algorithm shown in Figure 6-42. Use the `switch` statement to code the multiple-alternative selection structure. Enter the C++ instructions into a source file named TryThis11.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Save and run the program. Test the program using the following codes: 1, 2, 3, and 7. (The answers to TRY THIS Exercises are located at the end of the chapter.)

IPO chart information	C++ instructions
Input	
department code (1, 2, or 3)	`int deptCode = 0;`
Processing	
none	
Output	
salary	`int salary = 0;`

Algorithm

1. enter the department code
2. if (the department code is one of the following:)
   - 1 — salary = 25000
   - 2 — salary = 30000
   - 3 — salary = 32000
   - other — display "Invalid code"; salary = 0

   end if
3. display salary

Figure 6-42

MODIFY THIS

12. Complete TRY THIS Exercise 10. Enter (or copy) the instructions from the TryThis10.cpp file into a new source file named ModifyThis12.cpp. Be sure to change the filename in the first comment. Modify the code in the ModifyThis12.cpp file so that it verifies that the pay code is 1 through 10 only. If the pay code is not 1 through 10, don't prompt the user for the current pay. Instead, display the "Invalid pay code" message. Save and run the program. Test the program using 1 and 500 as the pay code and current pay. Now test it using the following six sets of input values: 4 and 450, 9 and 500, 2 and 625, 10 and 500, 6 and 150, and 11.

INTRODUCTORY

13. Karlton Learning wants a program that displays the amount of money a company owes for a seminar. The fee per person is based on the number of people the company registers, as shown in Figure 6-43. For example, if the company registers seven people, then the total amount owed is $560. If the user enters a number that is less than or equal to zero, the program should display an appropriate error message.

Number of registrants	Fee per person
1 through 4	$100
5 through 10	$80
11 or more	$60

Figure 6-43

a. Create an IPO chart for the problem, and then desk-check the algorithm five times, using the numbers 4, 8, 12, 0, and -2 as the number of people registered.

b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 6-42. Then code the algorithm into a program.

c. Desk-check the program using the same data used to desk-check the algorithm.

d. Enter your C++ instructions into a source file named Introductory13.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

e. Save and run the program. Test the program using the same data used to desk-check the program.

INTRODUCTORY

14. The owner of Harry's Car Sales pays each salesperson a commission based on his or her monthly sales. The sales ranges and corresponding commission rates are shown in Figure 6-44.

Monthly sales ($)	Commission rate
0 – 19,999.99	4%
20,000 – 29,999.99	5%
30,000 – 39,999.99	6%
40,000 – 49,999.99	7%
50,000 or more	9%
Less than 0	0%

Figure 6-44

a. Create an IPO chart for the problem, and then desk-check the algorithm six times, using sales of 2500, 28500.35, 35678.99, 42300, 50000, and -3.

b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 6-42. Then code the algorithm into a program.

c. Desk-check the program using the same data used to desk-check the algorithm.

d. Enter your C++ instructions into a source file named Introductory14.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Display the commission in fixed-point notation with two decimal places.

e. Save and run the program. Test the program using the same data used to desk-check the program.

INTERMEDIATE

15. In this exercise, you create a program that displays the number of daily calories needed to maintain your current weight. The number of calories is based on your gender, activity level, and weight. The formulas for calculating the daily calories are shown in Figure 6-45.

Formulas

Moderately active female: total daily calories = weight multiplied by 12 calories per pound

Relatively inactive female: total daily calories = weight multiplied by 10 calories per pound

Moderately active male: total daily calories = weight multiplied by 15 calories per pound

Relatively inactive male: total daily calories = weight multiplied by 13 calories per pound

Test data

Gender	Activity	Weight
F	I	150
F	A	120
M	I	180
M	A	200

Figure 6-45

a. Create an IPO chart for the problem, and then desk-check the algorithm four times, using the test data included in Figure 6-45.

b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 6-42. Then code the algorithm into a program.

c. Desk-check the program using the same data used to desk-check the algorithm.

d. Enter your C++ instructions into a source file named Intermediate15.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

e. Save and run the program. Test the program using the same data used to desk-check the program.

INTERMEDIATE

16. In this exercise, you create a program that adds, subtracts, multiplies, or divides two integers. The program will need to get a letter (A for addition, S for subtraction, M for multiplication, or D for division) and two integers from the user. If the user enters an invalid letter, the program should not ask the user for the two integers. Instead, it should display an appropriate error message before the program ends. If the letter is A (or a), the program should calculate and display the sum of both integers. If the letter is S (or s), the program should display the difference between both integers. When calculating the difference, always subtract the smaller number from the larger one. If the letter is M (or m), the program should display the product of both integers. If the letter is D (or d), the program should divide both integers, always dividing the larger number by the smaller one. Figure 6-46 shows the test data you will use for this exercise.

Test data

Operation	First integer	Second integer
A	10	20
a	45	15
S	65	50
s	7	13
G		
M	10	20
d	45	15
d	50	100

Figure 6-46

a. Create an IPO chart for the problem, and then desk-check the algorithm using the test data shown in Figure 6-46.

b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 6-42. Then code the algorithm into a program.

c. Desk-check the program using the same data used to desk-check the algorithm.

d. Enter your C++ instructions into a source file named Intermediate16.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

e. Save and run the program. Test the program using the same data used to desk-check the program.

ADVANCED

17. In this exercise, you create a program that converts US Dollars to a different currency. The currencies and exchange rates are listed in Figure 6-47. The user should be allowed to choose the currency. (Hint: Designate a code for each currency and use `cout` statements to display the code and corresponding currency.) The number of American Dollars should always be an integer that is greater than or equal to zero.

Currency	Exchange rate
Canada Dollar	1.01615
Eurozone Euro	.638490
India Rupee	40.1798
Japan Yen	104.390
Mexico Peso	10.4613
South Africa Rand	7.60310
United Kingdom Pound	.504285

Figure 6-47

a. Create an IPO chart for the problem, and then desk-check the algorithm nine times. For the first seven desk-checks, convert 10 American Dollars to each of the seven different currencies. For the eighth desk-check, test the algorithm using -3 as the number of American Dollars. For the last deck-check, test the algorithm using an invalid currency code.

b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 6-42. Then code the algorithm into a program.

c. Desk-check the program using the same data used to desk-check the algorithm.

d. Enter your C++ instructions into a source file named Advanced17.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Display the results in fixed-point notation with two decimal places.

e. Save and run the program. Test the program using the same data used to desk-check the program.

18. Shopper Haven wants a program that displays the number of reward points a customer earns each month. The reward points are based on the customer's membership type and total monthly purchase amount, as shown in Figure 6-48. ADVANCED

Membership type	Total monthly purchase ($)	Reward points
Standard	Less than 75	5% of the total monthly purchase
	75 – 149.99	7.5% of the total monthly purchase
	150 and over	10% of the total monthly purchase
Plus	Less than 150	6% of the total monthly purchase
	150 and over	13% of the total monthly purchase
Premium	Less than 200	4% of the total monthly purchase
	200 and over	15% of the total monthly purchase

Figure 6-48

a. Create an IPO chart for the problem, and then desk-check the algorithm appropriately.

b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 6-42. Then code the algorithm into a program.

c. Desk-check the program using the same data used to desk-check the algorithm.

d. Enter your C++ instructions into a source file named Advanced18.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Display the reward points in fixed-point notation with no decimal places.

e. Save and run the program. Test the program using the same data used to desk-check the program.

19. In this exercise, you experiment with the `switch` statement. ADVANCED

a. Follow the instructions for starting C++ and opening the Advanced19.cpp file. The file is contained in either the Cpp6\Chap06\Advanced19 Project folder or the Cpp6\Chap06 folder. Run the program. When you are prompted to enter a grade, type d and press Enter. What, if anything, did the `switch` statement display on the screen? Stop the program.

b. Enter a `break;` statement in the `case 'd':` clause. Save and then run the program. When you are prompted to enter a grade, type d and press Enter. What, if anything, did the `switch` statement display on the screen? Stop the program.

c. Remove the `break;` statement from the `case 'd':` clause. Also remove the `break;` statement from the `case 'f':` clause. Save and then run the program. When you are prompted to enter a grade, type d and press Enter. What, if anything, did the `switch` statement display on the screen? Stop the program.

d. Put the `break;` statement back in the `case 'f':` clause, and then save and run the program. When you are prompted to enter a grade, type d and press Enter. Stop the program.

ADVANCED

20. In this exercise, you experiment with the `switch` statement.

a. Follow the instructions for starting C++ and opening the Advanced20.cpp file. The file is contained in either the Cpp6\Chap06\Advanced20 Project folder or the Cpp6\Chap06 folder. The program uses the `switch` statement to display the names of the gifts mentioned in the song "The Twelve Days of Christmas."

b. Run the program. When you are prompted to enter the day, type the number 1 and press Enter. The names of the gifts for the first through the twelfth day appear on the screen. Stop the program.

c. Run the program again. When you are prompted to enter the day, type the number 9 and press Enter. The names of the gifts for the ninth through the twelfth day appear on the screen. Stop the program.

d. Modify the program so that it displays only the name of the gift corresponding to the day entered by the user. For example, when the user enters the number 4, the program should display the "4 calling birds" message only.

e. Save and then run the program. When you are prompted to enter the day, type the number 4 and press Enter. The "4 calling birds" message should appear on the screen. Stop the program, and then test it using the numbers 1 and 9.

ADVANCED

21. In this exercise, you will include a Boolean value in a `switch` statement. Follow the instructions for starting C++ and opening the Advanced21.cpp file. The file is contained in either the Cpp6\Chap06\Advanced21 Project folder or the Cpp6\Chap06 folder. Replace the dual-alternative `if` statement with a `switch` statement. Save and then run the program. Test the program appropriately.

SWAT THE BUGS

22. Follow the instructions for starting your C++ system and opening the SwatTheBugs22.cpp file. The file is contained in either the Cpp6\Chap06\SwatTheBugs22 Project folder or the Cpp6\Chap06 folder. Test the program using 1 as the code and 10 as the old price. Then test it using the following data: 2 and 10, 3 and 20, and 4. Debug the program.

Answers to TRY THIS Exercises



Pencil and Paper

1. See Figure 6-49.

```
if (monthNum == 1)
     cout << "January";
else if (monthNum == 2)
     cout << "February";
else if (monthNum == 3)
     cout << "March";
else if (monthNum == 4)
     cout << "April";
else if (monthNum == 5)
     cout << "May";
else if (monthNum == 6)
     cout << "June";
else if (monthNum == 7)
     cout << "July";
else if (monthNum == 8)
     cout << "August";
else if (monthNum == 9)
     cout << "September";
else if (monthNum == 10)
     cout << "October";
else if (monthNum == 11)
     cout << "November";
else if (monthNum ==12)
     cout << "December";
else
     cout << "Incorrect month number";
//end if
```

Figure 6-49

2. See Figure 6-50.

```
switch (code)
{
     case '1':
                rate = .02;
                break;
     case '2':
     case '3':
     case '4':
                rate = .05;
                break;
     case '5':
                rate = .1;
                break;
     case '6':
     case '7':
                rate = .15;
                break;
     default:
                rate = -1;;
}    //end switch
```

Figure 6-50



Computer

10. See Figures 6-51 and 6-52.

IPO chart information	C++ instructions
Input	
pay code	`int code = 0;`
current pay	`double currentPay = 0.0;`
raise rate 1 (3%)	`const double RATE1 = .03;`
raise rate 2 (4%)	`const double RATE2 = .04;`
raise rate 3 (5%)	`const double RATE3 = .05;`
Processing	
raise	`double raise = 0.0;`
Output	
new pay	`double newPay = 0.0;`
Algorithm	

1. enter the pay code and current pay	`cout << "Pay code: ";`
	`cin >> code;`
	`cout << "Current pay: ";`
	`cin >> currentPay;`

2. if (the pay code is 4 or 9)
calculate the raise by multiplying the current pay by raise rate 3
else if (the pay code is 2 or 10)
calculate the raise by multiplying the current pay by raise rate 2
else
calculate the raise by multiplying the current pay by raise rate 1
end if
	`if (code == 4 || code == 9)`
	`raise = currentPay * RATE3;`
	`else if (code == 2 || code == 10)`
	`raise = currentPay * RATE2;`
	`else`
	`raise = currentPay * RATE1;`
	`//end if`

3. calculate the new pay by adding the raise to the current pay	`newPay = currentPay + raise;`
4. display the new pay	`cout << "New pay: $" <<` `newPay << endl;`

Figure 6-51

```
 1  //TryThis10.cpp - displays the new pay
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <iomanip>
 6  using namespace std;
 7
 8  int main()
 9  {
10     //declare named constants and variables
11     const double RATE1 = .03;
12     const double RATE2 = .04;
13     const double RATE3 = .05;
14     int code           = 0;
15     double currentPay  = 0.0;
16     double raise       = 0.0;
17     double newPay      = 0.0;
18
19     //enter input items
20     cout << "Pay code: ";
21     cin >> code;
22     cout << "Current pay: ";
23     cin >> currentPay;
24
25     //calculate raise and new pay
26     if (code == 4 || code == 9)
27          raise = currentPay * RATE3;
28     else if (code == 2 || code == 10)
29          raise = currentPay * RATE2;
30     else
31          raise = currentPay * RATE1;
32     //end if
33     newPay = currentPay + raise;
34
35     //display new pay
36     cout << fixed << setprecision(2);
37     cout << "New pay: $" << newPay << endl;
38
39     system("pause");
40     return 0;
41 }   //end of main function
```

Figure 6-52

11. See Figure 6-53.

IPO chart information	C++ instructions
Input	
department code (1, 2, or 3)	`int deptCode = 0;`
Processing	
none	
Output	
salary	`int salary = 0;`
Algorithm	
1. enter the department code	`cout << "Department code (1, 2, or 3): ";` `cin >> deptCode;`
2. if (the department code is one of the following:)	`switch (deptCode)` `{`

1 salary = 25000	`case 1:`
	`  salary = 25000;`
	`  break;`

2 salary = 30000	`case 2:`
	`  salary = 30000;`
	`  break;`

3 salary = 32000	`case 3:`
	`  salary = 32000;`
	`  break;`

other display "Invalid code"
salary = 0
	`default:`
	`  cout << "Invalid code" << endl;`
	`  salary = 0;`

end if	`} //end switch`
3. display salary	`cout << "Salary: $" << salary << endl;`

Figure 6-53

CHAPTER 7

The Repetition Structure

After studying Chapter 7, you should be able to:

- Differentiate between a pretest loop and a posttest loop
- Include a pretest loop in pseudocode
- Include a pretest loop in a flowchart
- Code a pretest loop using the C++ `while` statement
- Utilize counter and accumulator variables
- Code a pretest loop using the C++ `for` statement

Repeating Program Instructions

Recall that all computer programs are written using one or more of three control structures: sequence, selection, and repetition. You learned about the sequence and selection structures in previous chapters. This chapter provides an introduction to the repetition structure. Programmers use the **repetition structure**, referred to more simply as a **loop**, when they need the computer to repeatedly process one or more program instructions. The loop contains a condition that controls whether the instructions are repeated. In many programming languages, the condition can be phrased in one of two ways. It can either specify the requirement for repeating the instructions or specify the requirement for *not* repeating them. The requirement for repeating the instructions is referred to as the **looping condition**, because it indicates when the computer should continue "looping" through the instructions. The requirement for *not* repeating the instructions is referred to as the **loop exit condition,** because it tells the computer when to exit (or stop) the loop. An example may help illustrate the difference between the looping condition and loop exit condition. You've probably heard the old adage "Make hay while the sun shines." The "while the sun shines" is the looping condition, because it tells you when to continue making hay. The adage also could be phrased as "Make hay until the sun is no longer shining." In this case, the "until the sun is no longer shining" is the loop exit condition, because it indicates when you should stop making hay. Every looping condition has an opposing loop exit condition; in other words, one is the opposite of the other. In the C++ programming language, the repetition structure's condition is always phrased as a looping condition, which means it always contains the requirement for repeating the instructions within the loop.

A repetition structure can be either a pretest loop or a posttest loop. In both types of loops, the condition is evaluated with each repetition (or iteration) of the loop. In a **pretest loop**, the condition is evaluated *before* the instructions within the loop are processed. In a **posttest loop**, the evaluation occurs *after* the instructions within the loop are processed. Depending on the result of the evaluation, the instructions in a pretest loop may never be processed. The instructions in a posttest loop, however, always will be processed at least once. Of the two types of loops, the pretest loop is the most commonly used. You will learn about pretest loops in this chapter; posttest loops are covered in Chapter 8.



Pretest and posttest loops also are called top-driven and bottom-driven loops, respectively.



You can nest loops, which means you can place one loop within another loop. Nested loops are covered in Chapter 8.

The programmer determines whether a problem's solution requires a repetition structure by studying the problem specification. The first problem specification you will examine in this chapter involves Robin, the mechanical woman. The problem specification and an illustration of the problem are shown in Figure 7-1, along with a correct algorithm written in pseudocode. The algorithm uses only the sequence structure, because no decisions need to be made and no instructions need to be repeated.

Robin is sitting at a table in a bookstore. Robin needs to sign a copy of her bestselling book on Robotics for a customer.

1. accept the book from the customer
2. place the book on the table
3. open the front cover of the book
4. sign your name on the first page
5. close the book
6. return the book to the customer
7. thank the customer

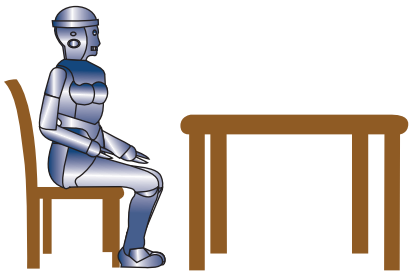


Figure 7-1 A problem that requires the sequence structure only

Now let's change the problem specification slightly. Here again, Robin is sitting at a table in the bookstore; but this time, she's there for a book signing. The store has just opened, and there's already a long line of customers. Robin doesn't want to disappoint her fans, so she plans on staying until every book is signed. Consider how this change will affect the original algorithm from Figure 7-1. The original algorithm contains the instructions for signing only one customer's book. In the modified algorithm, Robin will need to follow the same instructions for every customer waiting in line. Figure 7-2 shows the modified problem specification along with the modified algorithm, which contains both the sequence and repetition structures. The repetition structure begins with the *repeat while (there are customers in line)* clause and ends with the *end repeat* clause. The instructions between both clauses are indented to indicate that they are part of the repetition structure. The instructions between both clauses are called the **loop body**. The portion within parentheses in the *repeat while (there are customers in line)* clause specifies the repetition structure's condition. The condition is phrased as a looping condition, because it tells Robin when to repeat the instructions. In this case, she should repeat the instructions as long as (or while) "there are customers in line." Similar to the condition in a selection structure, the condition in a repetition structure must evaluate to a Boolean value: either true or false. The condition in Figure 7-2 evaluates to true when there are customers in line and evaluates to false when there are no customers in line. If the condition evaluates to true, Robin should follow the loop body instructions. She should skip over those instructions when the condition evaluates to false.

The opposite of *repeat while (there are customers in line)* is *repeat until (there are no customers in line)*.

Robin is sitting at a table in a bookstore, attending her book signing. Customers are standing in line waiting for Robin to sign their copy of her bestselling book on Robotics. Robin needs to sign each customer's book.

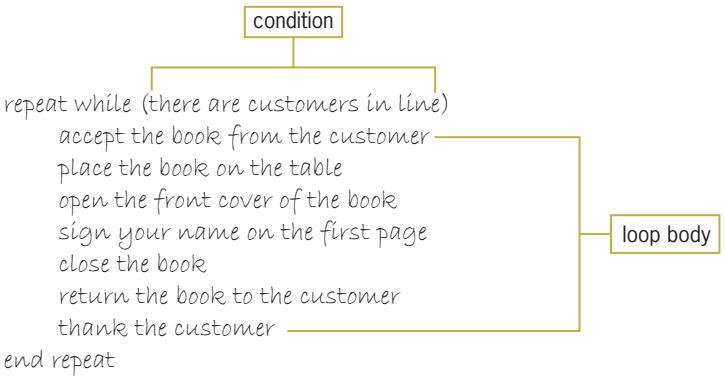

condition
repeat while (there are customers in line)
accept the book from the customer
place the book on the table
open the front cover of the book
sign your name on the first page
close the book
return the book to the customer
thank the customer
loop body
end repeat

Figure 7-2 A problem that requires the sequence and repetition structures

Using a Pretest Loop to Solve a Real-World Problem

As you already know, not every solution to a problem requires a loop. Consider, for example, the problem specification and IPO chart for Carroll Cabinets; both are shown in Figure 7-3. The problem specification indicates that the bonus for only one employee needs to be calculated and displayed. Therefore, the three instructions in the algorithm will need to be processed only once. Because no instructions need to be repeated, the algorithm does not require a loop.

Problem specification
At the beginning of each year, the president of Carroll Cabinets is paid a 5% bonus. The bonus is based on the amount of sales made by the company during the previous year. The payroll clerk wants a program that calculates and displays the bonus amount.

Input	Processing	Output
bonus rate (5%) sales	Processing items: none	bonus

Algorithm:
1. enter the sales
2. calculate the bonus by multiplying the sales by the bonus rate
3. display the bonus

calculates and displays the bonus for one employee only

Figure 7-3 Problem specification and IPO chart for the Carroll Cabinets program

Now consider the Miller Incorporated problem specification and IPO chart, which are shown in Figure 7-4. The problem specification is similar to the one for Carroll Cabinets, except it indicates that the program will need to calculate and display the bonus for more than one employee. You could use the algorithm shown earlier in Figure 7-3 to solve the Miller Incorporated problem. However, that algorithm is inefficient for the current problem, because it calculates and displays only one bonus amount. A program based on the algorithm in Figure 7-3 would need to be executed once for each salesperson receiving a bonus. In other words, if Miller Incorporated had 100 salespeople, the payroll clerk would need to run the program 100 times. The Miller Incorporated problem specification, you will notice, indicates that the payroll clerk wants to calculate and display each salesperson's bonus without having to run the program more than once. To accomplish this, the algorithm will need a repetition structure, as shown in Figure 7-4. After running a program based on the algorithm in Figure 7-4, the payroll clerk can calculate and display the bonus amount for any number of salespeople. The program will end when the payroll clerk enters -1 (a negative number one) as the sales amount.

Problem specification

In January of each year, Miller Incorporated pays a 5% bonus to each of its salespeople. The bonus is based on the amount of sales made by the salesperson during the previous year. The payroll clerk wants a program that calculates and displays the bonus amounts for as many salespeople as needed without having to run the program more than once. Because the sales amounts entered by the payroll clerk will always be positive numbers, the payroll clerk will indicate that he is finished with the program by entering a sales amount of –1 (a negative number 1).

Input	Processing	Output
bonus rate (5%) sales	Processing items: none	bonus

```
Algorithm:
1. enter the sales
2. repeat while (the sales are not equal to -1)
       calculate the bonus by multiplying the sales by
       the bonus rate
       display the bonus
       enter the sales
   end repeat
```

calculates and displays the bonus for as many salespeople as needed

Figure 7-4 Problem specification and IPO chart for the Miller Incorporated program



Sentinel values are often referred to as trip values, because they release the loop from its task. They also are called trailer values, because they are the last values entered before the loop ends.

Figure 7-5 identifies the important components of the algorithm shown in Figure 7-4. With very rare exceptions, every loop has a condition and a loop body. In a pretest loop, the condition appears at the beginning of the loop. As mentioned earlier, the condition must result in a Boolean value: either true or false. The condition in Figure 7-5 evaluates to true when the sales amount is not equal to -1 and evaluates to false when it *is* equal to -1. Some loops, such as the one shown in Figure 7-5, require the user to enter a special value to end the loop. Values that are used to end loops are referred to as **sentinel values**. In the loop in Figure 7-5, the sentinel value is -1. The sentinel value should be one that is easily distinguishable from the valid data recognized by the program. The number 1000 would not be a good sentinel value for the loop in Figure 7-5, because it is possible for a salesperson to sell $1,000 in product. The number -1, on the other hand, is a good sentinel value for the loop, because the problem specification states that the sales amount is always a positive number.

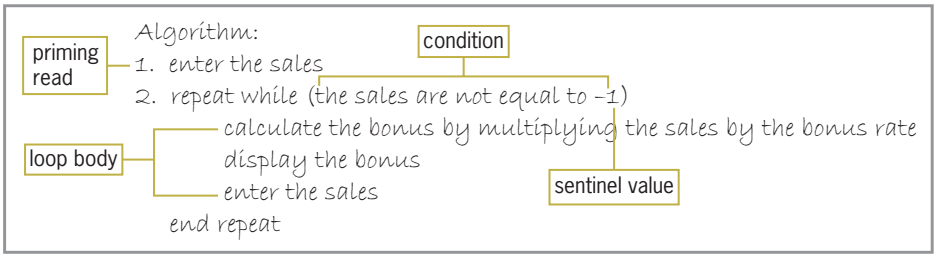


Figure 7-5 Components of the algorithm from Figure 7-4

When a loop's condition evaluates to true, the one or more instructions listed in the loop body are processed; otherwise, the loop body instructions are skipped over. After each processing of the loop body instructions, the loop's condition is reevaluated to determine whether the instructions should be processed again. The loop body instructions are processed and the loop's condition is evaluated until the condition evaluates to false, at which time the loop ends and processing continues with the instruction immediately following the end of the loop. Keep in mind that, because the condition in a pretest loop is evaluated *before* any of the instructions within the loop body are processed, it is possible that the loop body instructions may not be processed at all. For example, if the payroll clerk at Miller Incorporated enters the number -1 as the first sales amount, the condition in Figure 7-5's loop will evaluate to false and the instructions in the loop body will not be processed during that run of the program.

Notice that the algorithm in Figure 7-5 contains two *enter the sales* instructions. One of the instructions appears above the loop, and the other appears as the last instruction in the loop body. The *enter the sales* instruction above the loop is referred to as the **priming read**, because it is used to prime (prepare or set up) the loop. The priming read initializes the loop condition by providing its first value. In this case, the priming read gets only the first sales amount from the user. The first sales amount is compared to the sentinel value (-1) and determines whether the loop body instructions are processed at all. If the loop body instructions are processed, the *enter the sales* instruction in the loop body gets the remaining sales amounts (if any)

from the user. This instruction is referred to as the **update read**, because it allows the user to update the value of the input item (in this case, the sales amount) that controls the looping condition. The update read is often an exact copy of the priming read. Keep in mind that if you don't include the update read in the loop body, there will be no way to enter the sentinel value after the loop body instructions are processed the first time. This is because the priming read is processed only once and gets only the first sales amount from the user. You will learn more about this in the section titled "The `while` Statement" later in the chapter.

Flowcharting a Pretest Loop

Figure 7-6 shows the Miller Incorporated algorithm in flowchart form. Recall that the oval in a flowchart is the start/stop symbol, the rectangle is the process symbol, the parallelogram is the input/output symbol, and the diamond is the decision symbol. The diamond in Figure 7-6 indicates the beginning of a repetition structure (loop). Like the diamond in a selection structure, the diamond in a repetition structure contains a condition that evaluates to either true or false only. The condition determines whether the instructions within the loop body are processed. Also like the diamond in a selection structure, the diamond in a repetition structure has one flowline entering the symbol and two flowlines leaving the symbol. The two flowlines leading out of the diamond are marked with a "T" (for true) and an "F" (for false). The flowline marked with a "T" leads to the loop body, which contains the instructions to be processed when the loop's condition evaluates to true. The flowline marked with an "F", on the other hand, leads to the instructions to be processed when the loop's condition evaluates to false. Notice that the flowline entering the diamond, along with the diamond and the symbols and flowlines within the true path, form a circle or loop. It is this loop (circle) that distinguishes the repetition structure from the selection structure in a flowchart.

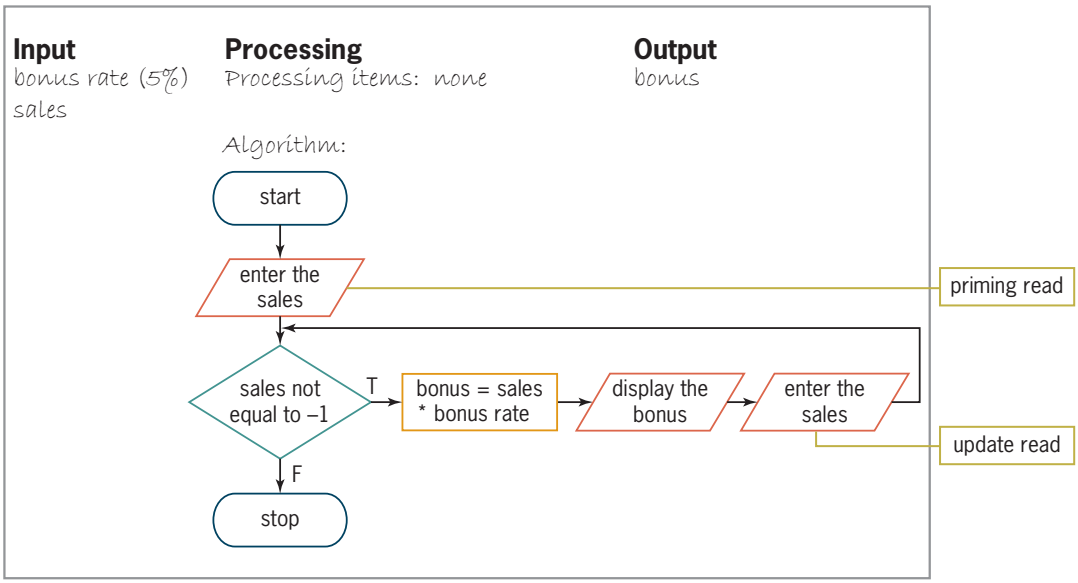


Figure 7-6 Miller Incorporated algorithm shown in flowchart form

To illustrate how a loop operates in a program, you will desk-check the algorithm in Figure 7-6 using the following sales data: 10000, 25000, and -1. The bonus amounts should be $500 and $1,250. First, you record the input and output items in a desk-check table, as shown in Figure 7-7.

bonus rate	sales	bonus
.05		

Figure 7-7 Input and output items entered in the desk-check table

Next, you follow each of the symbols in the flowchart, from top to bottom, recording in the desk-check table any changes made to the values of the input and output items. The first symbol is the start oval, which merely marks the beginning of the flowchart. The next symbol is a parallelogram that gets the first sales amount from the user. This symbol represents the priming read. Figure 7-8 shows the first sales amount recorded in the desk-check table.

bonus rate	sales	bonus
.05	10000	

Figure 7-8 First sales amount recorded in the desk-check table

The next symbol in the flowchart is a diamond that represents the condition in a pretest loop. You can tell that the loop is a pretest loop (rather than a posttest loop), because the diamond appears *before* the symbols in both the true and false paths. The loop's condition tells the computer to compare the sales amount entered by the user with the sentinel value, which is the number -1. In this case, the condition evaluates to true, because 10000 is not equal to -1. When the condition evaluates to true, the computer processes the instructions in the loop body. The first two instructions in the loop body calculate and display the bonus amount. Figure 7-9 shows the first salesperson's bonus information recorded in the desk-check table. The bonus amount agrees with your manually calculated results.

bonus rate	sales	bonus
.05	10000	500

Figure 7-9 First salesperson's bonus information recorded in the desk-check table

The last instruction in the loop body in Figure 7-6 is contained in a parallelogram. The instruction gets the next salesperson's sales amount—in this case, 25000—from the user. Recall that this instruction is the update read. After the user enters the sales amount, the loop's condition is reevaluated to determine whether the loop should be processed again (a true condition) or end (a false condition). Recall that the condition is contained in the diamond located at the top of the loop. In this case, the condition evaluates to true, because 25000 is not equal to -1. As a result, the bonus amount is calculated and then displayed on the screen. Figure 7-10 shows the second salesperson's information recorded in the desk-check table.

bonus rate	sales	bonus
.05	~~10000~~	~~500~~
	25000	1250

Figure 7-10 Second salesperson's information recorded in the desk-check table

The last parallelogram in the loop body gets the next salesperson's sales amount from the user. In this case, the user enters the sentinel value (-1) as the sales. Next, the loop's condition is reevaluated to determine whether the loop should be processed again (a true condition) or end (a false condition). In this case, the condition evaluates to false, because the sales amount entered by the user is equal to -1. When the loop's condition evaluates to false, the computer skips over the loop body instructions and processes the instruction immediately following the end of the loop. In Figure 7-6's flowchart, the stop oval follows the loop and marks the end of the flowchart. The completed desk-check table is shown in Figure 7-11.

bonus rate	sales	bonus
.05	~~10000~~	~~500~~
	~~25000~~	1250
	-1	

Figure 7-11 Completed desk-check table

You can code a pretest loop in C++ using either the `while` statement or the `for` statement. First, you will learn how to use the `while` statement. The `for` statement is covered later in the chapter.

The `while` Statement

Figure 7-12 shows the syntax of the `while` statement, which you can use to code a pretest loop in a C++ program. As the boldfaced text in the syntax indicates, the keyword `while` and the parentheses that surround the *condition* are essential components of the statement. The italicized items in the syntax indicate where the programmer must supply information. In this case, the programmer must supply the looping condition. The condition must be a Boolean expression, which is an expression that evaluates to either true or false. The expression can contain variables, constants, functions, arithmetic operators, comparison operators, and logical operators. Besides providing the condition, the programmer also must provide the loop body statements, which are the statements to be processed when the condition evaluates to true. If more than one statement needs to be processed, the statements must be entered as a statement block by enclosing them in a set of braces (`{}`). You also can include the braces when a loop body contains only one statement. By doing this, you won't need to remember to enter the braces when statements are added subsequently to the loop body. Forgetting to enter the braces is a common error made when typing the `while` statement in a program. Although not a requirement, it is a good programming practice to use a comment (such as `//end while`) to mark the end of the `while` statement. The comment will make your program easier to read and understand. Also included in Figure 7-12 are examples of using the `while` statement. In Example 1, the `while (age > 0)` clause tells the computer to repeat the

loop body instructions as long as (or while) the value in the `age` variable is greater than zero. The loop will stop when the user enters either the number 0 or a negative number. In Example 2, the `while (makeEntry == 'Y' || makeEntry == 'y')` clause indicates that the loop body instructions should be repeated as long as the value in the `makeEntry` variable is either the uppercase letter Y or the lowercase letter y. In this case, the loop will stop when the user enters anything other than the letter Y or the letter y. You also could write the `while` clause as either `while (toupper(makeEntry) == 'Y')` or `while (tolower(makeEntry) == 'y')`.

HOW TO Use the `while` Statement

Syntax

while (*condition***)**
either one statement or a statement block to be processed as long as the condition is true
`//end while`

Example 1

```
int age = 0;

cout << "Enter age: ";
cin >> age;
while (age > 0)
{
    cout << "You entered " << age << endl;
    cout << "Enter age: ";
    cin >> age;
}   //end while
```

Example 2

```
char makeEntry = ' ';
double sales = 0.0;

cout << "Enter a sales amount? (Y/N)";
cin >> makeEntry;
while (makeEntry == 'Y' || makeEntry == 'y')
{
    cout << "Enter the sales: ";
    cin >> sales;
    cout << "You entered " << sales << endl;
    cout << "Enter a sales amount? (Y/N)";
    cin >> makeEntry;
}   //end while
```

Figure 7-12 How to use the `while` statement

Figure 7-13 shows the IPO chart information and corresponding C++ instructions for the Miller Incorporated program. The first three statements in Figure 7-13 declare and initialize the `RATE` named constant and the `sales` and `bonus` variables. The `cout << "First sales amount (-1 to stop): ";` statement prompts the user to enter the first sales amount, and the `cin >> sales;` statement stores the user's response in the

`sales` variable. The looping condition in the `while (sales != -1)` clause compares the value stored in the `sales` variable with the sentinel value. If the `sales` variable does not contain the sentinel value, the looping condition evaluates to true and the loop body instructions are processed. Those instructions calculate and display the bonus, then prompt the user to enter the next sales amount, and then store the user's response in the `sales` variable. Each time the user enters a sales amount, the looping condition in the `while` clause compares the sales amount to the sentinel value. When the user enters the sentinel value—in this case, -1—as the sales amount, the looping condition evaluates to false. As a result, the loop body instructions are skipped over and processing continues with the instruction located immediately below the end of the loop. Keep in mind that if you forget to enter the update read (`cin >> sales;`) in the loop body, the computer will process the loop body instructions indefinitely. This is because, without the `cin >> sales;` statement in the loop body, there will be no way to change the value stored in the `sales` variable once the loop body instructions are processed. A loop whose instructions are processed indefinitely is referred to as either an **endless loop** or an **infinite loop**. Usually, you can stop a program that contains an endless loop by pressing Ctrl+c (press and hold down the Ctrl key as you tap the letter c, and then release both keys); you also can use the Command Prompt window's Close button. A sample run of the program is shown in Figure 7-14. (The program uses the `fixed` and `setprecision` stream manipulators to display the bonus amounts in fixed-point notation with two decimal places.)



You can practice stopping an endless loop by completing either Lab 7-5 or Computer Exercise 17.



The loop will stop when the sales amount is equal to -1, because "equal to" is the opposite of "not equal to."

IPO chart information	**C++ instructions**
Input	
bonus rate (5%)	`const double RATE = .05;`
sales	`int sales = 0;`
Processing	
none	
Output	
bonus	`double bonus = 0.0;`
Algorithm	
1. enter the sales	`cout << "First sales amount (-1 to stop): ";` `cin >> sales;`
2. repeat while (the sales are not equal to -1)	`while (sales != -1)`
	`{`
calculate the bonus by multiplying the sales by the bonus rate	`bonus = sales * RATE;`
display the bonus	`cout << "Bonus: $" << bonus;` `cout << endl << endl;`
enter the sales	`cout << "Next sales amount (-1 to stop): ";` `cin >> sales;`
end repeat	`} //end while`

Figure 7-13 IPO chart information and C++ instructions for the Miller Incorporated program

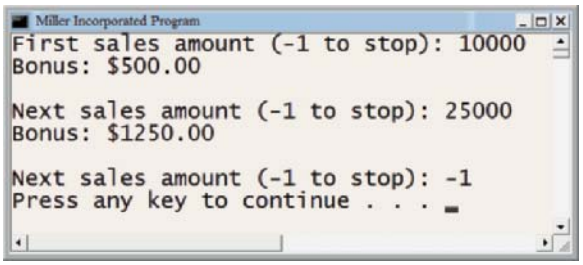

Miller Incorporated Program
First sales amount (-1 to stop): 10000
Bonus: $500.00
Next sales amount (-1 to stop): 25000
Bonus: $1250.00
Next sales amount (-1 to stop): -1
Press any key to continue . . .

Figure 7-14 A sample run of the Miller Incorporated program



The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 7-1

1. Write a C++ `while` clause that processes the loop body instructions as long as the value in the `quantity` variable is greater than the number 0.

2. Write a C++ `while` clause that stops the loop when the value in the `quantity` variable is less than the number 0. (Hint: Change the loop exit condition to a looping condition.)

3. Write a C++ `while` clause that processes the loop body instructions as long as the value in the `inStock` variable is greater than the value in the `reorder` variable.

4. Write a C++ `while` clause that processes the loop body instructions as long as the value in a `char` variable named `letter` is either Y or y.

5. Which of the following is a good sentinel value for a program that inputs the number of hours each employee worked during the week?

   a. -9

   b. 32

   c. 45.5

   d. 7

Using Counters and Accumulators

Some algorithms require you to calculate a subtotal, a total, or an average. You make these calculations using a repetition structure that includes a counter, an accumulator, or both. A **counter** is a numeric variable used for counting something, such as the number of employees paid in a week. An **accumulator** is a numeric variable used for accumulating (adding together) something, such as the total dollar amount of a week's payroll. Two tasks are associated with counters and accumulators: initializing and updating. **Initializing** means to assign a beginning value to the counter or accumulator. Typically, counters and accumulators are initialized to the number 0. However, they can be initialized to any number, depending on the value required by the algorithm. The initialization task is performed before the loop is processed, because it needs to be performed only once. **Updating**, often referred to as

incrementing, means adding a number to the value stored in the counter or accumulator. The number can be either positive or negative, integer or non-integer. A counter is always updated by a constant value—typically the number 1—whereas an accumulator is updated by a value that varies. The assignment statement that updates a counter or an accumulator is placed in the body of a loop, because the update task must be performed each time the loop body instructions are processed. The Sales Express program, which you view next, includes a counter, an accumulator, and a repetition structure.

The Sales Express Program

Figure 7-15 shows the problem specification, IPO chart information, and C++ instructions for the Sales Express program. (The flowchart for this program is contained in the Cpp6\Chap07\Ch7Flowcharts.pdf file.) The program's input is each salesperson's sales amount; its output is the average sales amount. The program uses two processing items: a counter and an accumulator. The counter (an `int` variable named `numSales`) keeps track of the number of sales amounts entered and is initialized to the number 0. The accumulator (a `double` variable named `totalSales`) keeps track of the total sales and is initialized to 0.0.

Problem specification
Sales Express wants a program that displays the average amount the company sold during the prior year. The sales manager will enter each salesperson's sales. The program will use a counter to keep track of the number of sales amounts entered and an accumulator to total the sales amounts. When the sales manager has finished entering the sales amounts, the program will calculate the average sales amount by dividing the value stored in the accumulator by the value stored in the counter. It then will display the average sales amount on the screen. The sales manager will indicate that she is finished with the program by entering a negative number as the sales amount. If the sales manager does not enter any sales amounts, the program should display the "No sales entered" message.

IPO chart information	**C++ instructions**
Input	
sales	`double sales = 0.0;`
Processing	
number of sales entered (counter)	`int numSales = 0;`
total sales (accumulator)	`double totalSales = 0.0;`
Output	
average sales	`double average = 0.0;`
Algorithm	
1. enter the sales	`cout << "First sales amount (negative number to stop): ";` `cin >> sales;`
2. repeat while (the sales are at least 0)	`while (sales >= 0.0)`
	`{`
add 1 to the number of sales entered	`numSales = numSales + 1;`
add the sales to the	`totalSales = totalSales +`



The loop will stop when the sales amount is less than 0.0, because "less than" is the opposite of "greater than or equal to."

Figure 7-15 Problem specification, IPO chart information, and C++ instructions for the Sales Express program *(continues)*

The *enter the sales* instruction located above the loop is the priming read, and the one within the loop is the update read.

(continued)

```
      total sales                        sales;

      enter the sales                    cout << "Next sales amount
                                         (negative number to stop): ";
                                         cin >> sales;
   end repeat                         } //end while

3. if (the number of sales entered is  if (numSales > 0)
   greater than 0)
                                      {
      calculate the average sales by     average = totalSales /
      dividing the total sales by the    numSales;
      number of sales entered

      display the average sales          cout << "Average: $" << average
                                         << endl;
                                      }
   else                               else
      display "No sales entered"         cout << "No sales entered"
      message                            << endl;
   end if                             //end if
```

Figure 7-15 Problem specification, IPO chart information, and C++ instructions for the Sales Express program

You can observe the way counters and accumulators are used in a program by desk-checking the code shown in Figure 7-15. You will do this using the following sales amounts: 30000, 40000, and -3. The average sales amount should be $35,000. After declaring and initializing the appropriate variables, the code prompts the user to enter the first sales amount and then stores the user's response in the `sales` variable. See Figure 7-16.

sales	numSales	totalSales	average
~~0.0~~	0	0.0	0.0
30000.0			

Figure 7-16 Desk-check table after the first sales amount is entered

The `while (sales >= 0.0)` clause begins a pretest loop that repeats the loop body instructions as long as (or while) the `sales` variable contains a value that is greater than or equal to 0.0. Notice that the condition compares the variable's value with 0.0 rather than with 0. This is because the values being compared should have the same data type—in this case, `double`. The loop in Figure 7-15 stops when the `sales` variable contains a sentinel value. For this loop, a sentinel value is any value that is less than 0.0. As the desk-check table in Figure 7-16 shows, the current value in the `sales` variable is greater than or equal to 0.0; therefore, the computer will process the instructions in the loop body. The first instruction updates the counter variable's value by adding the number 1 to it. The second instruction updates the accumulator variable's value by adding the current contents of the `sales` variable to it. You also can write the first two instructions in the loop

Unlike the loop in the Miller Incorporated program (shown earlier in Figure 7-13), the loop in the Sales Express program has more than one sentinel value.

body as `numSales += 1;` and `totalSales += sales;`. The desk-check table in Figure 7-17 shows the updated values assigned to the counter and accumulator variables.

sales	numSales	totalSales	average
~~0.0~~	~~0~~	~~0.0~~	0.0
30000.0	1	30000.0	

Figure 7-17 Desk-check table showing the first update to the counter and accumulator variables

The last two instructions in the loop body prompt the user to enter the next sales amount and then store the user's response—in this case, 40000—in the `sales` variable. Next, the condition in the `while (sales >= 0.0)` clause is reevaluated to determine whether the loop body instructions should be processed again (a true condition) or skipped over (a false condition). Here again, the loop's condition evaluates to true. As a result, the first instruction in the loop body updates the `numSales` variable's value by adding the number 1 to it. The second instruction updates the `totalSales` variable's value by adding the current contents of the `sales` variable to it. See Figure 7-18.

sales	numSales	totalSales	average
~~0.0~~	~~0~~	~~0.0~~	0.0
~~30000.0~~	~~1~~	~~30000.0~~	
40000.0	2	70000.0	

Figure 7-18 Desk-check table after the second update to the counter and accumulator variables

The last two instructions in the loop body prompt the user to enter the next sales amount and then store the user's response in the `sales` variable. In this case, the user enters the number -3, which is a sentinel value. Next, the condition in the `while (sales >= 0.0)` clause is reevaluated to determine whether the loop body instructions should be processed again or skipped over. In this case, the loop's condition evaluates to false, because the `sales` variable's value (-3.0) is not greater than or equal to 0.0. When the condition evaluates to false, the loop body instructions are skipped over and the loop ends; processing continues with the statement immediately following the loop. In the Sales Express program, an `if` statement follows the loop. The `if` statement's condition verifies that the value stored in the counter variable (`numSales`) is greater than the number 0, which is the variable's initial value. This verification is necessary because the first instruction in the `if` statement's true path uses the `numSales` variable as the divisor when calculating the average sales amount. Before using a variable as the divisor in an expression, you always should verify that the variable contains a value other than zero. This is because division by zero is not mathematically possible and will cause the program to end abruptly with an error. According to the desk-check table in Figure 7-18, the `numSales` variable contains the number 2. Therefore, the instructions in the `if` statement's true path first calculate the average sales amount (35000) and then display the amount on the screen before the program ends. Figure 7-19 shows the completed desk-check table for the Sales Express program, and Figures 7-20 and 7-21 show sample runs of the program. (The program uses the `fixed` and

`setprecision` stream manipulators to display the average sales amount in fixed-point notation with no decimal places.)

sales	numSales	totalSales	average
~~0.0~~	~~0~~	~~0.0~~	~~0.0~~
~~30000.0~~	~~1~~	~~30000.0~~	35000.0
~~40000.0~~	2	70000.0	
-3.0			

Figure 7-19 Completed desk-check table for the Sales Express program

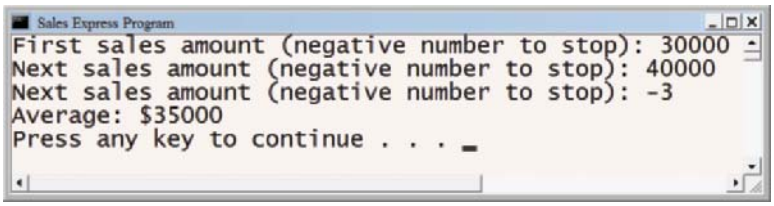

Sales Express Program
First sales amount (negative number to stop): 30000
Next sales amount (negative number to stop): 40000
Next sales amount (negative number to stop): -3
Average: $35000
Press any key to continue . . .

Figure 7-20 First sample run of the Sales Express program

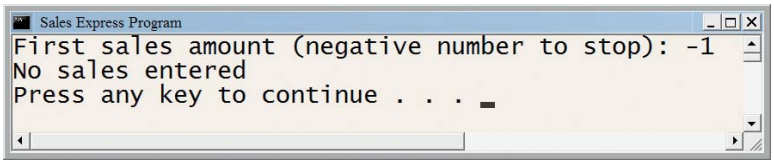


Figure 7-21 Second sample run of the Sales Express program



The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 7-2

1. Which of the following usually is updated by an amount that varies?
   a. accumulators
   b. counters
2. Write a C++ assignment statement that updates the `quantity` counter variable by 2.
3. Write a C++ assignment statement that updates the `total` counter variable by -3.
4. Write a C++ assignment statement that updates the `totalPurchases` accumulator variable by the value stored in the `purchases` variable.

Counter-Controlled Pretest Loops

In both the Miller Incorporated and Sales Express programs, the termination of the loop is determined by a sentinel value that is entered by the user at the keyboard. Other loops are controlled using a counter rather than a sentinel value; such loops are referred to as **counter-controlled loops**. The

Jasper Music Company program provides an example of a counter-controlled loop. The program's problem specification, IPO chart information, and C++ instructions are shown in Figure 7-22. (The program's flowchart is contained in the Cpp6\Chap07\Ch7Flowcharts.pdf file.) The input is the quarterly sales for each of the three regions, and the output is the total quarterly sales. The program uses one processing item: a counter that will keep track of the number of times the loop instructions are repeated. In this case, the loop instructions need to be repeated three times, once for each sales region. Notice in the C++ instructions that the `numRegions` counter variable is initialized to the number 1, which corresponds to the first sales region. It's also updated by 1 each time the loop instructions are processed.



The initialization and update of the counter variable in counter-controlled loops are comparable to the priming and update reads in loops controlled by a sentinel value.

Problem specification
The sales manager at Jasper Music Company wants a program that allows him to enter the quarterly sales amount made in each of three regions: Region 1, Region 2, and Region 3. The program should calculate the total quarterly sales and then display the result on the screen. The program will use a counter to ensure that the sales manager enters exactly three sales amounts. It will use an accumulator to total the sales amounts.

IPO chart information	C++ instructions
Input	
region's quarterly sales	`int regionSales = 0;`
Processing	
number of regions (counter: 1 to 3)	`int numRegions = 1;`
Output	
total quarterly sales (accumulator)	`int totalSales = 0;`
Algorithm	
1. repeat while (the number of regions is less than 4)	`while (numRegions < 4)`
	`{`
enter the region's quarterly sales	`cout << "Enter region " << numRegions << "'s quarterly sales: ";` `cin >> regionSales;`
add the region's quarterly sales to the total quarterly sales	`totalSales += regionSales;`
add 1 to the number of regions	`numRegions += 1;`
end repeat	`} //end while`
2. display the total quarterly sales	`cout << "Total quarterly sales: $" << totalSales << endl;`



The loop will stop when the number of regions is equal to 4, because "greater than or equal to" is the opposite of "less than."



The `while` clause in Figure 7-22 also could be written as `while (numRegions <= 3)`.

Figure 7-22 Problem specification, IPO chart information, and C++ instructions for the Jasper Music Company program

You can observe the way a counter is used to stop a loop by desk-checking the code shown in Figure 7-22. You will do this using the following three sales amounts: 2500, 6000, and 2000. The first three statements in the code create and initialize three variables. Figure 7-23 shows the desk-check table after these statements are processed.

regionSales	numRegions	totalSales
0	1	0

Figure 7-23 Desk-check table after the variable declaration statements are processed

The `while (numRegions < 4)` clause begins a pretest loop that repeats its instructions as long as (or while) the value in the `numRegions` counter variable is less than 4; the loop stops when the variable's value is 4. According to the desk-check table in Figure 7-23, the value stored in the `numRegions` variable is less than 4. As a result, the loop's condition evaluates to true and the statements in the loop body are processed. The first statement prompts the user to enter Region 1's quarterly sales. The second statement stores the user's response (in this case, 2500) in the `regionSales` variable. The third statement adds the value stored in the `regionSales` variable to the value stored in the `totalSales` accumulator variable; the sum of both values is 2500. The last statement in the loop body adds the number 1 to the value stored in the `numRegions` counter variable; the result is 2. You also can write the last two instructions in the loop body as `totalSales = totalSales + regionSales;` and `numRegions = numRegions + 1;`. Figure 7-24 shows the desk-check table after the loop body instructions are processed the first time.

regionSales	numRegions	totalSales
~~0~~	~~1~~	~~0~~
2500	2	2500

Figure 7-24 Results of processing the loop body instructions the first time

Next, the condition in the `while (numRegions < 4)` clause is reevaluated to determine whether the loop body instructions should be processed again (a true condition) or skipped over (a false condition). According to the desk-check table in Figure 7-24, the value stored in the `numRegions` variable is less than 4. As a result, the condition evaluates to true and the statements in the loop body are processed. The first two statements prompt the user to enter Region 2's quarterly sales and then store the user's response (in this case, 6000) in the `regionSales` variable. The next statement adds the value stored in the `regionSales` variable to the value stored in the `totalSales` accumulator variable; the sum of both values is 8500. The last statement in the loop body adds the number 1 to the value stored in the `numRegions` counter variable; the result is 3. Figure 7-25 shows the desk-check table after the loop body instructions are processed the second time.

regionSales	numRegions	totalSales
~~0~~	~~1~~	~~0~~
~~2500~~	~~2~~	~~2500~~
6000	3	8500

Figure 7-25 Results of processing the loop body instructions the second time

Next, the condition in the `while (numRegions < 4)` clause is reevaluated to determine whether the loop body instructions should be processed again or skipped over. According to the desk-check table in Figure 7-25, the value stored in the `numRegions` variable is less than 4. As a result, the condition evaluates to true and the statements in the loop body are processed. The first two statements prompt the user to enter Region 3's quarterly sales and then store the user's response (in this case, 2000) in the `regionSales` variable. The next statement adds the value stored in the `regionSales` variable to the value stored in the `totalSales` accumulator variable; the sum of both values is 10500. The last statement in the loop body adds the number 1 to the value stored in the `numRegions` counter variable; the result is 4. Figure 7-26 shows the desk-check table after the loop body instructions are processed the third (and last) time.

regionSales	numRegions	totalSales
0	~~1~~	~~0~~
~~2500~~	~~2~~	~~2500~~
~~6000~~	~~3~~	~~8500~~
2000	4	10500

Figure 7-26 Results of processing the loop body instructions the third time

Next, the condition in the `while (numRegions < 4)` clause is reevaluated to determine whether the loop body instructions should be processed again or skipped over. At this point, the loop's condition evaluates to false, because the value stored in the `numRegions` variable is not less than 4. When the condition evaluates to false, the instructions in the loop body are skipped over and the loop ends; processing continues with the statement located immediately below the loop. In the code shown earlier in Figure 7-22, the statement immediately below the loop displays the contents of the `totalSales` variable on the screen. Notice that the termination of the loop is controlled by the loop itself, rather than by the user at the keyboard. Figure 7-27 shows a sample run of the Jasper Music Company program.

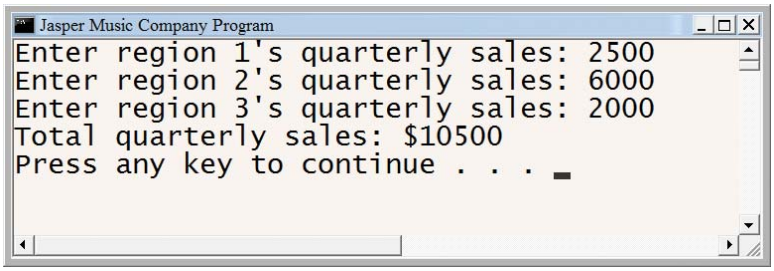

Jasper Music Company Program
Enter region 1's quarterly sales: 2500
Enter region 2's quarterly sales: 6000
Enter region 3's quarterly sales: 2000
Total quarterly sales: $10500
Press any key to continue . . .

Figure 7-27 A sample run of the Jasper Music Company program

The for Statement

Besides using the `while` statement, you also can use the `for` statement to code any pretest loop in C++. However, the most common use of the `for` statement is to code pretest loops whose processing is controlled by a counter—in other words, to code counter-controlled pretest loops. This is because the `for` statement

provides a more compact way of writing that type of loop. The `for` statement's syntax is shown in Figure 7-28. The statement begins with the `for` clause, which contains three arguments separated by two semicolons. As the brackets indicate, the first and third arguments are optional. In most `for` clauses, the *initialization* argument creates and initializes a counter variable that the computer uses to keep track of the number of times the loop body instructions are processed. The variable is local to the `for` statement, which means it can be used only within the statement's loop body. The variable will be removed from the computer's internal memory when the loop ends. The *condition* argument specifies the condition that must be true for the loop body instructions to be processed. The condition must be a Boolean expression, which is an expression that evaluates to either true or false. The expression can contain variables, constants, functions, arithmetic operators, comparison operators, and logical operators. The loop stops when its condition evaluates to false. The *update* argument in the `for` clause usually contains an expression that updates the counter variable specified in the *initialization* argument. Following the `for` clause is the body of the loop. The loop body contains the one or more statements that you want the computer to repeat. If the loop body contains more than one statement, the statements must be entered as a statement block by enclosing them in a set of braces ({}). However, you also can include the braces even when the loop body contains only one statement. Although it is not required by the C++ syntax, it is helpful to use a comment (such as `//end for`) to document the end of the `for` statement. The comment will make your program easier to read and understand. Also included in Figure 7-28 are examples of using the `for` statement. Example 1 displays the numbers 1, 2, and 3 on separate lines on the screen, whereas Example 2 displays the numbers 3, 2, and 1.

The *condition* argument in the `for` clause is a looping condition, because it specifies the requirement for repeating the loop instructions.

The *condition* in Example 1 in Figure 7-28 also could be written as *x* <= *3*. The condition in Example 2 also could be written as `x >= 1`.

HOW TO Use the `for` Statement

Syntax

for ([*initialization*]**;** *condition*; [*update*]**)**
    either one statement or a statement block to be processed as long as the condition is true
`//end for`

semicolons

Example 1: displays the numbers 1, 2, and 3 on separate lines on the screen

```
for (int x = 1; x < 4; x += 1)
    cout << x << endl;
//end for
```

you also can use `x = x + 1`

Example 2: displays the numbers 3, 2, and 1 on separate lines on the screen

```
for (int x = 3; x > 0; x = x - 1)
    cout << x << endl;
//end for
```

you also can use `x -= 1`

A common error made when typing the `for` clause is to use commas rather than semicolons.

Figure 7-28 How to use the `for` statement

Figure 7-29 describes the way the computer processes the code shown in Example 1 in Figure 7-28. Notice that the `for` statement in the example ends when the `x` variable contains the number 4.

Processing steps for Example 1

1. The *initialization* argument (`int x = 1`) tells the computer to create a variable named x and initialize it to the number 1.
2. The *condition* argument (`x < 4`) tells the computer to check whether the x variable's value is less than 4. It is, so the computer processes the statement in the loop body. That statement displays the x variable's value (1) on the screen.
3. The *update* argument (`x += 1`) tells the computer to add the number 1 to the contents of the x variable, giving 2.
4. The *condition* argument tells the computer to check whether the x variable's value is less than 4. It is, so the computer processes the statement in the loop body. That statement displays the x variable's value (2) on the screen.
5. The *update* argument tells the computer to add the number 1 to the contents of the x variable, giving 3.
6. The *condition* argument tells the computer to check whether the x variable's value is less than 4. It is, so the computer processes the statement in the loop body. That statement displays the x variable's value (3) on the screen.
7. The *update* argument tells the computer to add the number 1 to the contents of the x variable, giving 4.
8. The *condition* argument tells the computer to check whether the x variable's value is less than 4. It's not, so the computer stops processing the `for` loop and removes its local x variable. Processing continues with the statement following the end of the loop.

Figure 7-29 Processing steps for the code shown in Example 1 in Figure 7-28

In the remaining sections in this chapter, you will view three programs that use the `for` statement.

The Holmes Supply Program

Figure 7-30 shows the problem specification, IPO chart information, and C++ instructions for the Holmes Supply Company program.

Problem specification

The payroll manager at Holmes Supply Company wants a program that allows her to enter the payroll amount for each of three stores: Store 1, Store 2, and Store 3. The program should calculate the total payroll and then display the result on the screen. The program will use a counter to ensure that the payroll manager enters exactly three payroll amounts. It will use an accumulator to total the amounts.

IPO chart information	C++ instructions
Input	
store's payroll	`int storePayroll = 0;`
Processing	
number of stores (counter: 1 to 3)	this variable is created and initialized in the for clause
Output	
total payroll (accumulator)	`int totalPayroll = 0;`

Figure 7-30 Problem specification, IPO chart information, and C++ instructions for the Holmes Supply Company program *(continues)*

(continued)



The loop will stop when the value in the `numStores` variable is greater than 3, because "greater than" is the opposite of "less than or equal to."

Algorithm

```
1. repeat for (number of stores
   from 1 to 3)
       enter the store's payroll
       add the store's payroll to
       the total payroll
   end repeat
2. display the total payroll
```

```
for (int numStores = 1;
num Stores <= 3; numStores += 1)
{
   cout << "Store " << numStores
   << " payroll: ";
   cin >> storePayroll;
   totalPayroll += storePayroll;

} //end for
cout << "Total payroll: $"
<< totalPayroll << endl;
```

Figure 7-30 Problem specification, IPO chart information, and C++ instructions for the Holmes Supply Company program



The *condition* argument in Figure 7-30 also could be written as `numStores < 4`.

Desk-checking the code shown in Figure 7-30 will help you understand how the `for` statement works. You will desk-check the code using the following three payroll amounts: 15000, 25000, and 60000. First, the code declares and initializes two `int` variables named `storePayroll` and `totalPayroll`. The `for` clause in the `for` statement is processed next. The clause's *initialization* argument creates an `int` variable named `numStores` and initializes the variable to the number 1. The *initialization* argument is processed only once, at the beginning of the loop. Figure 7-31 shows the desk-check table after the declaration statements and *initialization* argument have been processed.

storePayroll	totalPayroll	numStores
0	0	1

Figure 7-31 Results of processing the declaration statements and *initialization* argument



The *condition* argument in the `for` clause is a looping condition, because it indicates when the loop instructions should be processed.

Next, the `for` clause's *condition* argument is evaluated to determine whether the loop body instructions should be processed (a true condition) or skipped over (a false condition). Notice that, like the condition in a `while` statement, the condition in a `for` statement is evaluated *before* the loop body instructions are processed. In this case, the loop body instructions will be processed only when the value in the `numStores` variable is less than or equal to the number 3. According to the desk-check table in Figure 7-31, the `numStores` variable contains the number 1. Therefore, the loop's condition evaluates to true and the statements in the loop body are processed. The first statement prompts the user to enter the amount of Store 1's payroll, and the second statement stores the user's response (in this case, 15000) in the `storePayroll` variable. The last statement in the loop body adds the value stored in the `storePayroll` variable to the value stored in the `totalPayroll` accumulator variable; the sum of both values is 15000. The `for` clause's *update* argument is processed next. The *update* argument adds the number 1 to the value stored in the `numStores` variable, giving 2. Figure 7-32 shows the desk-check table after the *update* argument is processed the first time.

storePayroll	totalPayroll	numStores
~~0~~	~~0~~	~~1~~
15000	15000	2

Figure 7-32 Desk-check table after the *update* argument is processed the first time

Next, the `for` clause's *condition* argument is reevaluated to determine whether the loop body instructions should be processed again or skipped over. Unlike the *initialization* argument, which is processed only once, the *condition* argument is processed with each repetition (or iteration) of the loop. According to the desk-check table in Figure 7-32, the `numStores` variable contains the number 2. Therefore, the `numStores <= 3` condition evaluates to true, and the statements in the loop body are processed again. The first statement prompts the user to enter Store 2's payroll, and the second statement stores the user's response (in this case, 25000) in the `storePayroll` variable. The last statement in the loop body adds the store's payroll to the total payroll; the sum of both values is 40000. The `for` clause's *update* argument is processed next. Like the *condition* argument, the *update* argument is processed with each repetition of the loop. The *update* argument adds the number 1 to the value stored in the `numStores` variable, giving 3. Figure 7-33 shows the desk-check table after the *update* argument is processed the second time.

storePayroll	totalPayroll	numStores
~~0~~	~~0~~	~~1~~
~~15000~~	~~15000~~	~~2~~
25000	40000	3

Figure 7-33 Desk-check table after the *update* argument is processed the second time

Next, the `for` clause's *condition* argument is reevaluated to determine whether the loop body instructions should be processed again or skipped over. According to the desk-check table in Figure 7-33, the `numStores` variable contains the number 3. Therefore, the `numStores <= 3` condition evaluates to true, and the statements in the loop body are processed again. The first statement prompts the user to enter Store 3's payroll, and the second statement stores the user's response (in this case, 60000) in the `storePayroll` variable. The last statement in the loop body adds the store's payroll to the total payroll; the sum of both values is 100,000. The `for` clause's *update* argument is processed next. The *update* argument adds the number 1 to the value stored in the `numStores` variable, giving 4. Figure 7-34 shows the desk-check table after the *update* argument is processed the third (and last) time.

storePayroll	totalPayroll	numStores
~~0~~	~~0~~	~~1~~
~~15000~~	~~15000~~	~~2~~
~~25000~~	~~40000~~	~~3~~
60000	100000	4

Figure 7-34 Desk-check table after the *update* argument is processed the third time

Next, the `for` clause's *condition* argument is reevaluated to determine whether the loop body instructions should be processed again or skipped over. At this point, the `numStores <= 3` condition evaluates to false, because the `numStores` variable contains the number 4. When the *condition* argument evaluates to false, the instructions in the loop body are skipped over and the loop ends. As a result, the computer removes the `for` statement's local variable, `numStores`, from internal memory. Processing continues with the instruction located immediately below the end of the loop. Notice that the loop stops when the `numStores` variable contains the number 4. Figure 7-35 shows a sample run of the Holmes Supply program.

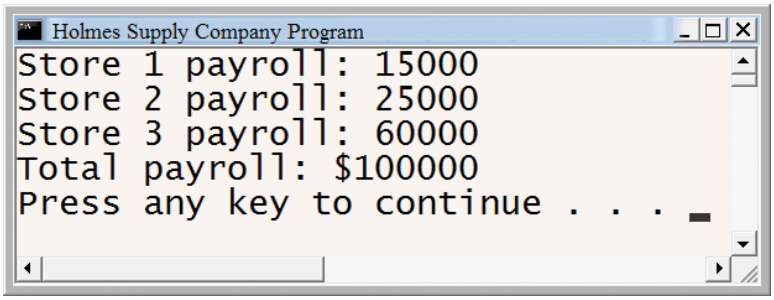

Holmes Supply Company Program
Store 1 payroll: 15000
Store 2 payroll: 25000
Store 3 payroll: 60000
Total payroll: $100000
Press any key to continue . . .

Figure 7-35 A sample run of the Holmes Supply Company program

The Colfax Sales Program

Figure 7-36 shows the problem specification, IPO chart information, and C++ instructions for the Colfax Sales program.

Problem specification
The sales manager at Colfax Sales wants a program that allows him to enter a sales amount. The program should calculate and display the appropriate commission using rates of 10%, 15%, 20%, and 25%. The program will use a counter to keep track of the four rates.

IPO chart information	**C++ instructions**
Input	
sales amount	`double sales = 0.0;`
Processing	
rate (counter: 10% to 25% in increments of 5%)	this variable is created and initialized in the for clause
Output	
commission	`double commission = 0.0;`
Algorithm	
1. enter the sales amount	`cout << "Enter the sales: ";` `cin >> sales;`

Figure 7-36 Problem specification, IPO chart information, and C++ instructions for the Colfax Sales program *(continues)*

(continued)

```
2. repeat for (rate from 10%          for (double rate = .1;
   to 25% in increment of 5%)         rate <= .25; rate = rate + .05)
                                      {
      calculate the commission           commission = sales * rate;
      by multiplying the sales
      amount by the rate

      display the commission             cout << rate * 100 <<
                                         "% commission: $" <<
                                         commission << endl;
   end repeat                         } //end for
```

Figure 7-36 Problem specification, IPO chart information, and C++ instructions for the Colfax Sales program

Figure 7-37 lists the steps the computer follows when processing the code shown in Figure 7-36, using a sales amount of $25,000. Notice that the `for` loop ends when the value stored in the `rate` variable is .3. Figure 7-38 shows a sample run of the Colfax Sales program.

Processing steps

1. The computer creates the `sales` and `commission` variables and initializes them to 0.0.
2. The computer prompts the user to enter a sales amount and then stores the user's response (in this case, 25000) in the `sales` variable.
3. The computer processes the `for` clause's *initialization* argument (`double rate = .1`), which creates the `rate` variable and initializes it to .1.
4. The computer processes the `for` clause's *condition* argument (`rate <= .25`), which checks whether the `rate` variable's value is less than or equal to .25. It is, so the computer processes the statements in the loop body. Those statements calculate a 10% commission and display the result (2500) on the screen.
5. The computer processes the `for` clause's *update* argument (`rate = rate + .05`), which adds the number .05 to the value stored in the `rate` variable; the result is .15.
6. The computer processes the `for` clause's *condition* argument, which checks whether the `rate` variable's value is less than or equal to .25. It is, so the computer processes the statements in the loop body. Those statements calculate a 15% commission and display the result (3750) on the screen.
7. The computer processes the `for` clause's *update* argument, which adds the number .05 to the value stored in the `rate` variable; the result is .2.
8. The computer processes the `for` clause's *condition* argument, which checks whether the `rate` variable's value is less than or equal to .25. It is, so the computer processes the statements in the loop body. Those statements calculate a 20% commission and display the result (5000) on the screen.
9. The computer processes the `for` clause's *update* argument, which adds the number .05 to the value stored in the `rate` variable; the result is .25.
10. The computer processes the `for` clause's *condition* argument, which checks whether the `rate` variable's value is less than or equal to .25. It is, so the computer processes the statements in the loop body. Those statements calculate a 25% commission and display the result (6250) on the screen.
11. The computer processes the `for` clause's *update* argument, which adds the number .05 to the value stored in the `rate` variable; the result is .3.

Figure 7-37 Processing steps for the code shown in Figure 7-36 *(continues)*

(continued)

12. The computer processes the `for` clause's *condition* argument, which checks whether the `rate` variable's value is less than or equal to .25. It's not, so the computer stops processing the `for` loop and removes its local `rate` variable from internal memory. Processing continues with the statement following the end of the loop.

Figure 7-37 Processing steps for the code shown in Figure 7-36

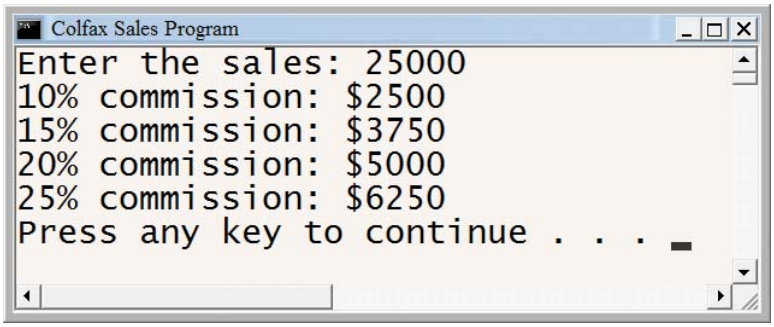


Figure 7-38 A sample run of the Colfax Sales program



Some programmers use the hexagon to represent any counter-controlled loop, even those coded with the `while` statement.

Many programmers use a hexagon, which is a six-sided figure, to represent the `for` clause in a flowchart. The hexagon contains four items, as shown in Figure 7-39. The four items are the name of the counter variable (`rate`), the variable's initial value (.1), the value used to update the variable (.05), and the last value for which the condition will evaluate to true (.25). Notice that a less than or equal sign (<=) precedes the .25 in the hexagon in Figure 7-39. The <= sign indicates that the loop body instructions will be processed as long as the `rate` variable's value is less than or equal to .25.

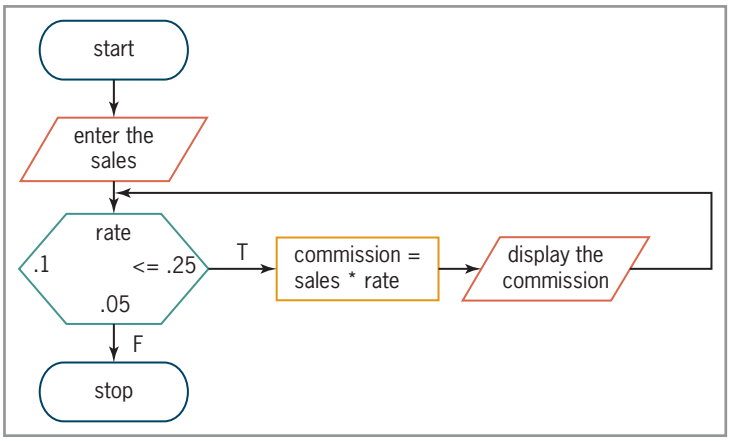


Figure 7-39 Colfax Sales algorithm shown in flowchart form

Another Version of the Miller Incorporated Program

Earlier in the chapter—more specifically, in Figure 7-13—you viewed the IPO chart information and C++ instructions for the Miller Incorporated

program. As you may remember, the program contains a pretest loop that is coded using the `while` statement. The loop allows the user to calculate and display the bonus amount for as many salespeople as needed without having to run the program again. Rather than use the `while` statement to code the pretest loop, you can use the `for` statement, as shown in Figure 7-40. The modifications made to the original code shown earlier in Figure 7-13 are shaded in Figure 7-40. Although the `for` statement is more commonly used to code counter-controlled loops, it can be used to code any pretest loop in C++. In this case, the `for` statement is used to code a pretest loop that is controlled by the user at the keyboard instead of by a counter. Notice that the `for` clause in Figure 7-40 contains only the *condition* argument. Although the *initialization* and *update* arguments are omitted from the `for` clause, the semicolons after the *initialization* and *condition* arguments must be included. Whether you use the `for` statement or the `while` statement to code the loop in Figure 7-40 is a matter of personal preference.

IPO chart information	**C++ instructions**
Input	
bonus rate (5%)	`const double RATE = .05;`
sales	`int sales = 0;`
Processing	
none	
Output	
bonus	`double bonus = 0.0;`
Algorithm	
1. enter the sales	`cout << "First sales amount (-1 to stop): ";` `cin >> sales;`
2. repeat while (the sales are not equal to -1)	`for (; sales != -1;)` `{`
calculate the bonus by multiplying the sales by the bonus rate	`bonus = sales * RATE;`
display the bonus	`cout << "Bonus: $" << bonus;` `cout << endl << endl;`
enter the sales	`cout << "Next sales amount (-1 to stop): ";` `cin >> sales;`
end repeat	`}  //end for`

semicolon after the *initialization* argument

condition argument

semicolon after the *condition* argument

Figure 7-40 IPO chart information and modified C++ instructions for the Miller Incorporated program

Figure 7-41 lists the steps the computer follows when processing the code shown in Figure 7-40, using sales amounts of $10000, $25000, and -1.



A sample run of the Miller Incorporated program is shown earlier in Figure 7-14.



The loop in Figure 7-40 will stop when the `sales` variable's value is equal to -1, because "equal to" is the opposite of "not equal to."

Processing steps

1. The computer creates the `RATE` named constant and initializes it to .05.
2. The computer creates the `sales` and `bonus` variables and initializes them to 0 and 0.0, respectively.
3. The computer prompts the user to enter the first sales amount and then stores the user's response (in this case, 10000) in the `sales` variable.
4. The computer processes the `for` clause's *condition* argument (`sales != -1`), which checks whether the `sales` variable's value is not equal to -1. The condition evaluates to true, so the computer processes the statements in the loop body. Those statements calculate and display the bonus (500). They also prompt the user to enter the next sales amount and store the user's response (in this case, 25000) in the `sales` variable.
5. The computer processes the `for` clause's *condition* argument, which checks whether the `sales` variable's value is not equal to -1. The condition evaluates to true, so the computer processes the statements in the loop body. Those statements calculate and display the bonus (1250). They also prompt the user to enter the next sales amount and store the user's response (in this case, -1) in the `sales` variable.
6. The computer processes the `for` clause's *condition* argument, which checks whether the `sales` variable's value is not equal to -1. In this case, the condition evaluates to false, so the computer stops processing the `for` loop. Processing continues with the statement following the end of the loop.

Figure 7-41 Processing steps for the code shown in Figure 7-40



The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 7-3

1. A program declares an `int` variable named `evenNum` and initializes it to 2. Write a C++ `while` loop that uses the `evenNum` variable to display the even integers between 1 and 9.

2. Which of the following `for` clauses processes the loop instructions as long as the `x` variable's value is less than or equal to the number 100?

   a. `for (int x = 10; x <= 100; x = x + 10)`

   b. `for (int x = 10, x <= 100, x = x + 10)`

   c. `for (int x == 10; x <= 100; x = x + 10)`

   d. `for (int x = x + 10; x <= 100; x = 10)`

3. The computer will stop processing the loop associated with the `for` clause from Question 2 when the `x` variable contains the number _____.

   a. 100

   b. 111

   c. 101

   d. 110

4. Write a `for` clause that processes the loop instructions as long as the value stored in the `x` variable is greater than the number 0. The `x` variable should be an `int` variable. Initialize the variable to the number 25 and update it by -5 with each repetition of the loop.

5. The computer will stop processing the loop associated with the `for` clause from Question 4 when the `x` variable contains the number _____.

6. Write a `for` statement that displays the even integers between 2 and 9 (inclusive) on the screen. Use `num` as the name of the counter variable.



LAB 7-1 Stop and Analyze

Study the program shown in Figure 7-42, and then answer the questions. The program calculates the average outside temperature.



The answers to the labs are located in Appendix A.

```
 1  //Lab7-1.cpp - calculates the average temperature
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <iomanip>
 6  using namespace std;
 7
 8  int main()
 9  {
10     //declare variables
11     int numberOfTemps = 0; //counter
12     int totalTemp     = 0; //accumulator
13     int temp          = 0;
14     double average    = 0.0;
15
16     //get first temperature
17     cout << "First temperature (999 to stop): ";
18     cin >> temp;
19
20     while (temp != 999)
21     {
22        //update counter and accumulator
23        numberOfTemps += 1;
24        totalTemp += temp;
25
26        //get remaining temperatures
27        cout << "Next temperature (999 to stop): ";
28        cin >> temp;
29     }  //end while
30
31     //verify that at least one temperature was entered
32     if (numberOfTemps > 0)
```

Figure 7-42 Code for Lab 7-1 *(continues)*

(continued)

```
33     {
34        //calculate and display average temperature
35        average = static_cast<double>(totalTemp) /
36             static_cast<double>(numberOfTemps);
37        cout << fixed << setprecision(1);
38        cout << endl << "Average temperature: "
39             << average << endl;
40     }
41     else
42        cout << "No temperatures were entered." << endl;
43     //end if
44
45     system("pause");
46     return 0;
47  }  //end of main function
```

your C++ development tool may not require this statement

Figure 7-42 Code for Lab 7-1

QUESTIONS

1. What are the program's input, processing, and output items?
2. Why do you think the number 999 was chosen as the sentinel value in the `while` clause on Line 20? Would a negative number be a good sentinel value for this program? Why or why not?
3. Why is the selection structure on Lines 32 through 43 necessary?
4. Are the type casts in the statement on Lines 35 and 36 necessary? Why or why not?
5. What is the purpose of the statement on Line 37?
6. Why was the `numberOfTemps` counter variable initialized to 0 rather than to 1?
7. Desk-check the program using the following temperatures and sentinel value: 78, 85, 67, and 999. What is the average temperature?
8. Follow the instructions for starting C++ and opening the Lab7-1.cpp file. Run the program and then enter the temperatures and sentinel value from Question 7. What does the program display as the average temperature? (The program's output should agree with the results of your desk-check from Question 7.)
9. Run the program again. Enter the sentinel value. What does the program display?
10. Change the `while` statement to a `for` statement. Save and then run the program. Enter the sentinel value. What does the program display?
11. Run the program again. Enter the following temperatures and sentinel value: -3, 32, -10, 40, and 999. What is the average temperature?



LAB 7-2 Plan and Create

In this lab, you will plan and create an algorithm for Professor Chang. The problem specification and example calculations are shown in Figure 7-43.

Problem specification
Professor Chang wants a program that allows him to enter a student's project and test scores. The professor assigns three projects and two tests. Each project is worth 50 points, and each test is worth 100 points. The program should calculate and display the total points the student earned on the projects and tests. It also should display the student's grade, which is based on the total points earned. Shown below is the grading scale that Professor Chang uses when assigning grades.

Total points earned	Grade
315 – 350	A
280 – 314	B
245 – 279	C
210 – 244	D
below 210	F

Example 1
Project and test scores: 45, 40, 41, 96, 89
Total points earned and grade: 311, B

Example 2
Project and test scores: 40, 35, 37, 73, 68
Total points earned and grade: 253, C

Figure 7-43 Problem specification and calculation examples for Lab 7-2

First, analyze the problem, looking for the output first and then for the input. In this case, Professor Chang wants the program to display the total points a student earned and his or her grade. To calculate the total points earned, the computer will need to know the student's project and test scores. The scores are the problem's input items and will be entered by Professor Chang. To determine the grade, the computer will need to know the professor's grading scale. The grading scale is provided in the problem specification. Next, plan the algorithm. Recall that most algorithms begin with an instruction to enter the input items into the computer, followed by instructions that process the input items, typically including the items in one or more calculations. Most algorithms end with one or more instructions that display, print, or store the output items. Figure 7-44 shows the completed IPO chart for the Professor Chang problem.

Input	Processing	Output
score (5 of them)	Processing items: none	total points earned grade

```
Algorithm:
1. enter the first score
2. repeat while (the score is not -1)
       add the score to the total points earned
       enter the next score
   end repeat
3. if (the total points earned >= 315)
       assign A as the grade
   else if (the total points earned >= 280)
       assign B as the grade
   else if (the total points earned >= 245)
       assign C as the grade
   else if (the total points earned >= 210)
       assign D as the grade
   else
       assign F as the grade
   end if
4. display the total points earned and the grade
```

Figure 7-44 Completed IPO chart for the Professor Chang problem

After completing the IPO chart, you then move on to the third step in the problem-solving process, which is to desk-check the algorithm. You will desk-check the algorithm in Figure 7-44 two times. For the first desk-check, you will use the following five scores and sentinel value: 45, 40, 41, 96, 89, and -1. You will use the following five scores and sentinel value for the second desk-check: 40, 35, 37, 73, 68, and -1. Figure 7-45 shows the completed desk-check table. Notice that the values in the total points earned and grade columns agree with the results of the manual calculations shown in Figure 7-43.

	score	total points earned	grade
first desk-check	~~45~~	~~45~~	
	~~40~~	~~85~~	
	~~41~~	~~126~~	
	~~96~~	~~222~~	
	~~89~~	311	
	-1		B
second desk-check	~~40~~	~~40~~	
	~~35~~	~~75~~	
	~~37~~	~~112~~	
	~~73~~	~~185~~	
	~~68~~	253	
	-1		C

Figure 7-45 Completed desk-check table for the Professor Chang algorithm

The fourth step in the problem-solving process is to code the algorithm into a program. You begin by declaring memory locations that will store the values of the input, processing (if any), and output items. The Professor Chang problem will need three memory locations to store the input and output items. The input item (score) will be stored in a variable, because the user should be allowed to change its value during runtime. The output items (total points earned and grade) also will be stored in variables, because their values will change based on the current value of the input item. The score and total points earned will always be integers, so you will store both in `int` variables. You will store the grade in a `char` variable. Figure 7-46 shows the IPO chart information and corresponding C++ instructions.

IPO chart information	C++ instructions
Input	
score (5 of them)	`int score = 0;`
Processing	
none	
Output	
total points earned	`int totalPoints = 0;`
grade	`char grade = ' ';`
Algorithm	
1. enter the first score	`cout << "First score (-1 to stop): ";` `cin >> score;`
2. repeat while (the score is not -1)	`while (score != -1)` `{`
add the score to the total points earned	`totalPoints += score;`
enter the next score	`cout << "Next score (-1 to stop): ";` `cin >> score;`
end repeat	`} //end while`
3. if (the total points earned >= 315)	`if (totalPoints >= 315)`
assign A as the grade	`grade = 'A';`
else if (the total points earned >= 280)	`else if (totalPoints >= 280)`
assign B as the grade	`grade = 'B';`
else if (the total points earned >= 245)	`else if (totalPoints >= 245)`
assign C as the grade	`grade = 'C';`
else if (the total points earned >= 210)	`else if (totalPoints >= 210)`
assign D as the grade	`grade = 'D';`
else	`else`
assign F as the grade	`grade = 'F';`
end if	`//end if`
4. display the total points earned and the grade	`cout << "Total points earned: " << totalPoints << endl;` `cout << "Grade: " << grade << endl;`

Figure 7-46 IPO chart information and C++ instructions for the Professor Chang program

The fifth step in the problem-solving process is to desk-check the program. You begin by placing the names of the declared variables and named constants (if any) in a new desk-check table, along with their initial values. You then desk-check the remaining C++ instructions in order, recording in the desk-check table any changes made to the variables. Figure 7-47 shows the completed desk-check table for the Professor Chang program. The results agree with those shown in the algorithm's desk-check table in Figure 7-45.

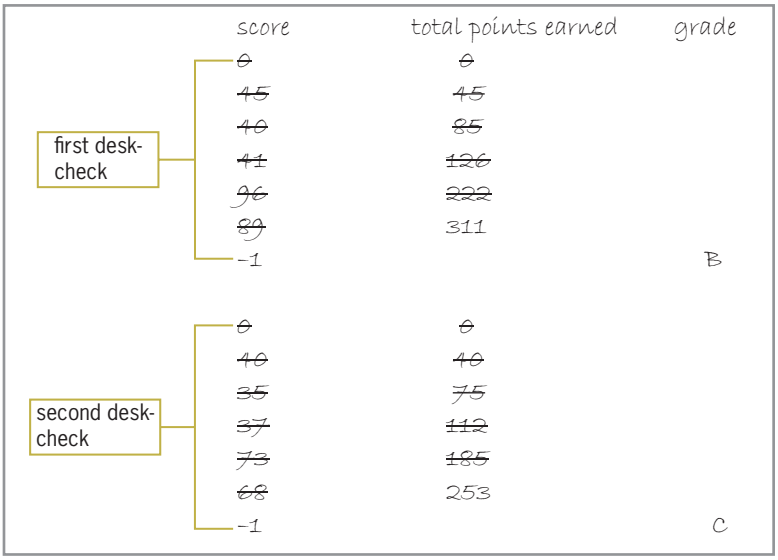


Figure 7-47 Completed desk-check table for the Professor Chang program

The final step in the problem-solving process is to evaluate and modify (if necessary) the program. Recall that you evaluate a program by entering its instructions into the computer and then using the computer to run (execute) it. While the program is running, you enter the same sample data used when desk-checking the program.

DIRECTIONS

Follow the instructions for starting your C++ development tool. Depending on the development tool you are using, you may need to create a new project; if so, name the project Lab7-2 Project and save it in the Cpp6\Chap07 folder. Enter the instructions shown in Figure 7-48 in a source file named Lab7-2.cpp. (Do not enter the line numbers.) Save the file in either the project folder or the Cpp6\Chap07 folder. Now follow the appropriate instructions for running the Lab7-2.cpp file. Use the sample data from Figure 7-47 to test the program. If necessary, correct any bugs (errors) in the program.

```
1  //Lab7-2.cpp - displays the total points earned and grade
2  //Created/revised by <your name> on <current date>
3
4  #include <iostream>
5
6  using namespace std;
7
8  int main()
```

Figure 7-48 Professor Chang program *(continues)*

(continued)

```
 9  {
10     //declare variables
11     int score       = 0;
12     int totalPoints = 0;  //accumulator
13     char grade      = ' ';
14
15     //get first score
16     cout << "First score (-1 to stop): ";
17     cin >> score;
18
19     while (score != -1)
20     {
21        //update accumulator, then get another score
22        totalPoints += score;
23        cout << "Next score (-1 to stop): ";
24        cin >> score;
25     } //end while
26
27     //determine grade
28     if (totalPoints >= 315)
29         grade = 'A';
30     else if (totalPoints >= 280)
31         grade = 'B';
32     else if (totalPoints >= 245)
33         grade = 'C';
34     else if (totalPoints >= 210)
35         grade = 'D';
36     else
37         grade = 'F';
38     //end if
39
40     //display the total points and grade
41     cout << "Total points earned: " << totalPoints << endl;
42     cout << "Grade: " << grade << endl;
43
44     system("pause");
45     return 0;
46  }  //end of main function
```

your C++ development tool may not require this statement

Figure 7-48 Professor Chang program



You also can write the statement on Line 22 in Figure 7-48 as `totalPoints = totalPoints + score;`.



LAB 7-3 Modify

If necessary, create a new project named Lab7-3 Project. Enter (or copy) the Lab7-2.cpp instructions into a new source file named Lab7-3.cpp. Change Lab7-2.cpp in the first comment to Lab7-3.cpp. Professor Chang now wants the program to display the total number of scores he enters. Modify the program appropriately, and then save and run the program. Test the program using the following five scores and

sentinel value: 45, 40, 41, 96, 89, and -1. The total points earned and grade should be 311 and B, respectively. In addition, the program should indicate that Professor Chang entered 5 scores. Now test the program a second time using the following scores and sentinel value: 25, 500 (Professor Chang inadvertently enters 500 rather than 50), 38, -500 (Professor Chang corrects his mistake by entering a negative number 500), 50, 64, 78, and -1. Does the program display the correct total points earned and grade? How many scores does the program indicate that Professor Chang entered?



LAB 7-4 Desk-Check

The code shown in Figure 7-49 should display the squares of the numbers from 1 through 5. In other words, it should display the numbers 1, 4, 9, 16, and 25. Desk-check the code. Did your desk-check reveal any errors in the code? If so, correct the code and then desk-check it again.

```
//declare variables
int squaredNumber = 0;

for (int number = 1; number < 5; number = number + 1)
{
     squaredNumber = number * number;
     cout << squaredNumber << endl;
}    //end for
```

Figure 7-49 Code for Lab 7-4



LAB 7-5 Debug

Follow the instructions for starting C++ and opening the Lab7-5.cpp file. The file is contained in either the Cpp6\Chap07\Lab7-5 Project folder or the Cpp6\Chap07 folder. Run the program. When you are prompted to enter a price, type 15.45 and press Enter. The "Next price:" prompt appears over and over again in the Command Prompt window, as shown in Figure 7-50. This is a result of the computer repeatedly processing the `cout << "Next price: ";` statement, which is contained in the body of the `while` loop, and it indicates that the program contains an endless (or infinite) loop. You can stop an endless loop by pressing Ctrl+c (press and hold down the Ctrl key as you tap the letter c, and then release both keys). (If you are using Microsoft Visual C++, you may need to click the Continue button after pressing Ctrl+c). Or, you can use the Close button on the Command Prompt window's title bar. Use either method to stop the endless loop, and then debug the program.



If you are using Microsoft Visual C++, you also can stop an endless loop by clicking Debug on the menu bar in the IDE, and then clicking Stop Debugging.

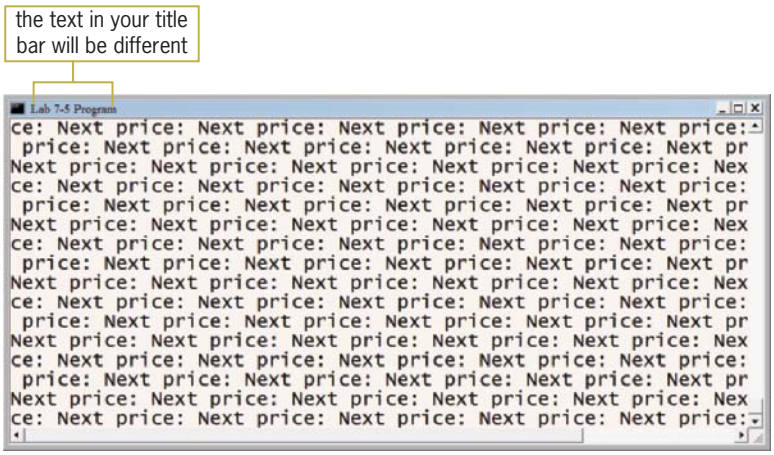


Figure 7-50 Command Prompt window showing that the program is in an endless loop

Summary

- You use the repetition structure, also called a loop, when you need the computer to repeatedly process one or more program instructions while the looping condition is true (or until the loop exit condition has been met).
- A repetition structure can be either a pretest loop or a posttest loop. In a pretest loop, the loop condition is evaluated *before* the instructions within the loop are processed. In a posttest loop, the evaluation occurs *after* the instructions within the loop are processed. Of the two types of loops, the pretest loop is the most commonly used.
- The condition appears at the beginning of a pretest loop and determines whether the instructions within the loop, referred to as the loop body, are processed. The loop's condition must result in either a true or false answer only. When the condition evaluates to true, the instructions listed in the loop body are processed; otherwise, the loop body instructions are skipped over.
- Some loops require the user to enter a special value, called a sentinel value, to end the loop. You should use a sentinel value that is easily distinguishable from the valid data recognized by the program. Other loops are terminated through the use of a counter.
- The input instruction that appears above the pretest loop's condition is referred to as the priming read, because it is used to prime (prepare or set up) the loop. The priming read gets only the first value from the user. The input instruction that appears within the loop gets the remaining values (if any) and is referred to as the update read.
- In most flowcharts, a diamond is used to represent a repetition structure's condition. The diamond is called the decision symbol.
- Counters and accumulators are used within a repetition structure to calculate subtotals, totals, and averages. All counters and accumulators must be initialized and updated. Counters are updated by a constant value, whereas accumulators are updated by an amount that varies.

- Many programmers use a hexagon to represent the `for` clause in a `for` statement.
- You can use either the `while` statement or the `for` statement to code a pretest loop in C++.

Key Terms

Accumulator—a numeric variable used for accumulating (adding together) something

Counter—a numeric variable used for counting something

Counter-controlled loops—loops whose processing and termination are controlled by a counter variable

Endless loop—a loop whose instructions are processed indefinitely; also called an infinite loop

Incrementing—another name for updating

Infinite loop—another name for an endless loop

Initializing—the process of assigning a beginning value to a memory location, such as a counter or accumulator variable

Loop—another name for the repetition structure

Loop body—the instructions within a loop

Loop exit condition—the requirement that must be met for the computer to stop processing the loop body instructions

Looping condition—the requirement that must be met for the computer to continue processing the loop body instructions

Posttest loop— a loop whose condition is evaluated *after* the instructions in its loop body are processed

Pretest loop—a loop whose condition is evaluated *before* the instructions in its loop body are processed

Priming read—the input instruction that appears above a loop; used to get the first input item from the user

Repetition structure—the control structure used to repeatedly process one or more program instructions; also called a loop

Sentinel values—values that are used to end loops; also called trip values or trailer values

Update read—the input instruction that appears within a loop and is associated with the priming read

Updating—the process of adding a number to the value stored in a counter or accumulator variable; also called incrementing

Review Questions

Refer to Figure 7-51 to answer Review Questions 1 through 4.

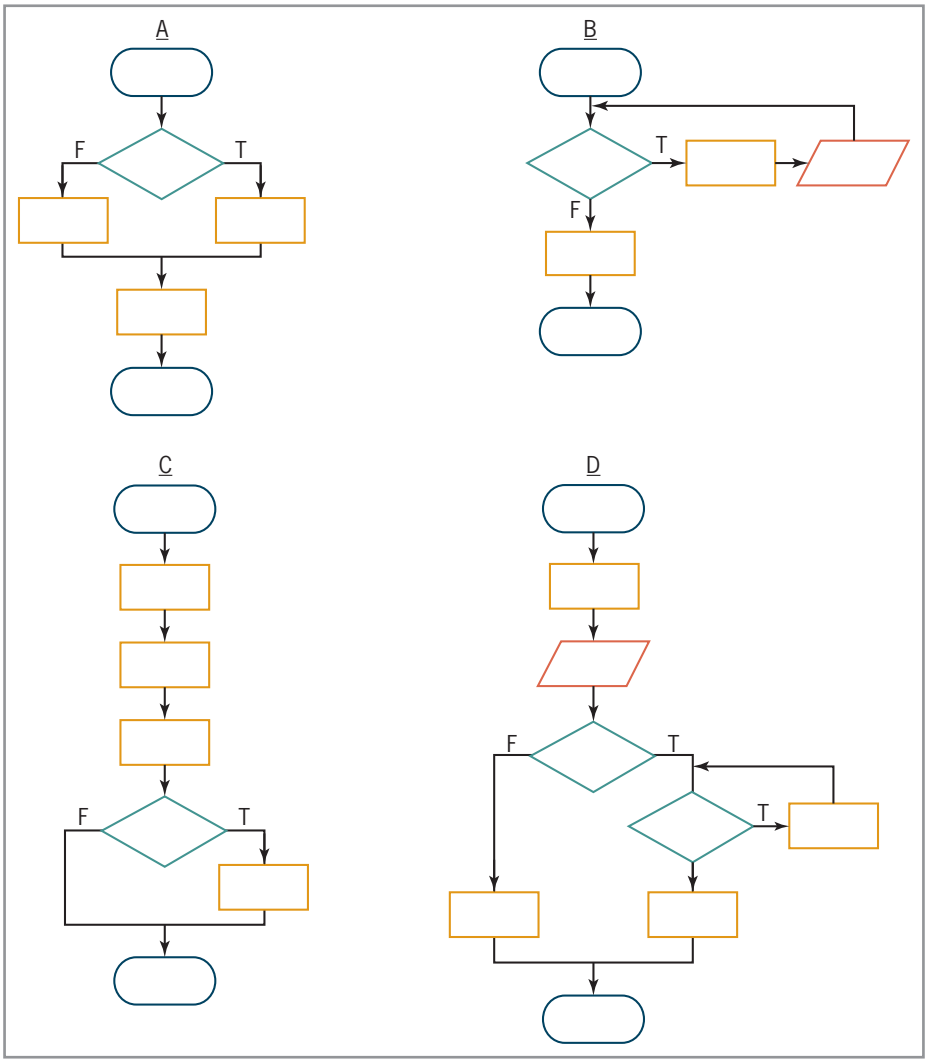


Figure 7-51

1. Which of the following control structures are used in flowchart A in Figure 7-51? (Select all that apply.)
   a. sequence
   b. selection
   c. repetition

2. Which of the following control structures are used in flowchart B in Figure 7-51? (Select all that apply.)
   a. sequence
   b. selection
   c. repetition

3. Which of the following control structures are used in flowchart C in Figure 7-51? (Select all that apply.)

   a. sequence

   b. selection

   c. repetition

4. Which of the following control structures are used in flowchart D in Figure 7-51? (Select all that apply.)

   a. sequence

   b. selection

   c. repetition

5. Which of the following `while` clauses tells the computer to exit the loop when the value in the `age` variable is less than the number 0?

   a. `while (age < 0)`

   b. `while age >= 0;`

   c. `while (age !> 0)`

   d. `while (age >= 0)`

6. Which of the following is a good sentinel value for a program that allows the user to enter a person's age?

   a. -4

   b. 350

   c. 999

   d. all of the above

7. Values that are used to end loops are referred to as _____ values.

   a. closing

   b. ending

   c. sentinel

   d. stop

8. A program allows the user to enter one or more numbers. The first input instruction will get the first number only and is referred to as the _____ read.

   a. entering

   b. initializer

   c. initializing

   d. priming

9. How many times will the computer process the `cout << numTimes << endl;` statement in the following code?

```
int numTimes = 0;
while (numTimes > 3)
{
     cout << numTimes << endl;
     numTimes = numTimes + 1;
}    //end while
```

a. 0

b. 1

c. 3

d. 4

10. How many times will the computer process the `cout << numTimes << endl;` statement in the following code?

```
for (int numTimes = 1; numTimes < 6; numTimes = numTimes + 1)
     cout << numTimes << endl;
//end for
```

a. 0

b. 1

c. 5

d. 6

11. What value in the `numTimes` variable stops the loop in Review Question 10?

a. 1

b. 5

c. 6

d. 7

12. How many times will the computer process the `cout << numTimes << endl;` statement in the following code?

```
for (int numTimes = 4; numTimes <= 10; numTimes += 2)
     cout << numTimes << endl;
//end for
```

a. 0

b. 3

c. 4

d. 12

13. What value in the `numTimes` variable stops the loop in Review Question 12?

    a. 4

    b. 6

    c. 10

    d. 12

14. Which of the following updates the `total` accumulator variable by the value in the `sales` variable?

    a. `total = total + sales;`

    b. `total = sales + total;`

    c. `total += sales`

    d. all of the above

15. Which of the following statements can be used to code a loop whose instructions you want processed 10 times?

    a. `for`

    b. `repeat`

    c. `while`

    d. either a or c

Exercises



Pencil and Paper

TRY THIS

1. Complete a desk-check table for the code shown in Figure 7-52. What will the code display on the computer screen? What `temp` variable value stops the loop? (The answers to TRY THIS Exercises are located at the end of the chapter.)

```
int temp = 0;
while (temp < 5)
{
     cout << temp << endl;
     temp = temp + 1;
}    //end while
```

Figure 7-52

TRY THIS

2. Complete a desk-check table for the code shown in Figure 7-53. What will the code display on the computer screen? What `num` variable value stops the loop? (The answers to TRY THIS Exercises are located at the end of the chapter.)

```
for (int num = 0; num <= 5; num += 2)
{
     cout << "Number: ";
     cout << num << endl;
}    //end for
```

Figure 7-53

3. Rewrite the code shown in Figure 7-52 to use the `for` statement. MODIFY THIS

4. Rewrite the code shown in Figure 7-53 to use the `while` statement. MODIFY THIS

5. Complete a desk-check table for the code shown in Figure 7-54. What will the code display on the computer screen? What `totalEmployee` variable value stops the loop? INTRODUCTORY

```
int totalEmployee = 0;
while (totalEmployee <= 5)
{
     cout << totalEmployee << endl;
     totalEmployee = totalEmployee + 2;
}    //end while
```

Figure 7-54

6. Write an assignment statement that updates a counter variable named `numStudents` by 1. INTRODUCTORY

7. Write an assignment statement that updates an accumulator variable named `totalPay` by the value in the `grossPay` variable. INTRODUCTORY

8. Write a C++ `while` clause that processes the loop instructions as long as the value in the `quantity` variable is greater than the number 100. INTRODUCTORY

9. Write a C++ `for` clause that processes the loop instructions 10 times. Use `numTimes` as the counter variable's name. INTRODUCTORY

10. Figure 7-28 in the chapter showed two examples of the `for` statement. List the processing steps for the code shown in Example 2. (Use Figure 7-29 as a guide.) INTRODUCTORY

11. Write a C++ `while` clause that stops the loop when the value in the `inStock` variable is less than or equal to the value in the `reorder` variable. INTERMEDIATE

12. Write an assignment statement that updates a counter variable named `quantity` by -5. INTERMEDIATE

INTERMEDIATE

13. Write an assignment statement that subtracts the contents of the `salesReturns` variable from the `sales` accumulator variable.

ADVANCED

14. Write two versions of the code to display the numbers 10 through 1 on the computer screen. In the first version, use the `for` statement. In the second version, use the `while` statement.

SWAT THE BUGS

15. The code shown in Figure 7-55 should display the numbers 1, 2, 3, and 4 on the computer screen. However, the code is not working correctly. Correct the errors in the code.

```
int num = 1;
while (num < 5)
     cout << num << endl;
//end while
```

Figure 7-55

SWAT THE BUGS

16. The code shown in Figure 7-56 should display each salesperson's commission. The commission is calculated by multiplying the salesperson's sales by 10%. The code is not working correctly. Correct the errors in the code.

```
double sales = 0.0;
double commission = 0.0;
cout << "Enter a sales amount: ";
cin >> sales;
while (sales > 0.0)
{
     commission = sales * .1;
     cout << commission << endl;
}    //end while
```

Figure 7-56



Computer

TRY THIS

17. In this exercise, you learn two ways to stop a program that is in an endless (infinite) loop.

    a. Follow the instructions for starting C++ and opening the TryThis17.cpp file. The file is contained in either the Cpp6\Chap07\TryThis17 Project folder or the Cpp6\Chap07 folder.

    b. Run the program. You can tell that the program is in an endless loop because it displays the number 0 over and over again in the Command Prompt window. In most cases, you can stop an endless loop by pressing Ctrl+c (press and hold down the Ctrl key as you tap the letter c, and then release both keys). Use the Ctrl+c key combination to stop the program. If you are using Microsoft



If you are using Microsoft Visual C++, you also can stop an endless loop by clicking Debug on the menu bar in the IDE, and then clicking Stop Debugging.

Visual C++, you may need to click the Continue button. If necessary, close the Command Prompt window.

c. Run the program again. You also can use the Command Prompt window's Close button to stop an endless loop. Click the Command Prompt window's Close button.

18. TRY THIS

Follow the instructions for starting C++ and opening the TryThis18.cpp file. The file is contained in either the Cpp6\Chap07\TryThis18 Project folder or the Cpp6\Chap07 folder. Complete the program by entering a `while` clause that processes the loop instructions when the user enters a number that is at least 0. Save and then run the program. Test the program using the following numbers: 4, 10, 0, and -3. (The answers to TRY THIS Exercises are located at the end of the chapter.)

19. TRY THIS

Follow the instructions for starting C++ and opening the TryThis19.cpp file. The file is contained in either the Cpp6\Chap07\TryThis19 Project folder or the Cpp6\Chap07 folder. Complete the program by entering a `while` clause that stops the loop when the user enters the letter N in either uppercase or lowercase. Save and then run the program. Test the program using the following characters: a, 4, $, and n. (The answers to TRY THIS Exercises are located at the end of the chapter.)

20. MODIFY THIS

Follow the instructions for starting C++ and opening the ModifyThis20.cpp file. The file is contained in either the Cpp6\Chap07\ModifyThis20 Project folder or the Cpp6\Chap07 folder. Change the `while` statement to a `for` statement. Save and then run the program. The word "Hello" should appear on the screen 10 times.

21. MODIFY THIS

In this exercise, you modify the program from Lab7-2.

a. If necessary, create a new project named ModifyThis21 Project. Enter (or copy) the Lab7-2.cpp instructions into a new source file named ModifyThis21.cpp. Change Lab7-2.cpp in the first comment to ModifyThis21.cpp.

b. If Professor Chang enters a score that is greater than 100 points, the program should ask him whether the score is correct. The program should not add a score that is more than 100 points to the accumulator without the professor's permission. Modify the program appropriately.

c. Save and then run the program. Enter 35, 45, and 50 as the first three scores, and then enter 105. When you are asked whether the 105 score is correct, respond that it is not correct. Now enter 100, 90, and -1. The program should display 320 as the total points earned and A as the grade.

d. Run the program again. Enter 27, 15, and 30 as the first three scores, and then enter 105. When you are asked whether the 105 score is correct, respond that it is correct. Now enter 70 and -1. The program should display 247 as the total points earned and C as the grade.

INTRODUCTORY

22. Follow the instructions for starting C++ and opening the Introductory22.cpp file. The file is contained in either the Cpp6\Chap07\Introductory22 Project folder or the Cpp6\Chap07 folder. Complete the program by entering a `while` clause that processes the loop instructions as long as the user enters the letter Y in either uppercase or lowercase. Save and then run the program. Test the program using the following three sets of data: y and 100, Y and 200, and n. The total sales amount should be $300. Run the program again. When you are asked whether you want to enter a sales amount, type N and press Enter. Notice that you were not asked to enter the sales amount. Explain why.

INTRODUCTORY

23. Figure 7-57 shows the flowchart for an algorithm that displays the numbers 20, 40, 60, 80, 100, 120, 140, 160, and 180 on the screen. Code the algorithm into a program; use the `while` statement. The counter variable should be an `int` variable named `number`. Then enter your C++ instructions into a source file named Introductory23.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Save and then run the program.

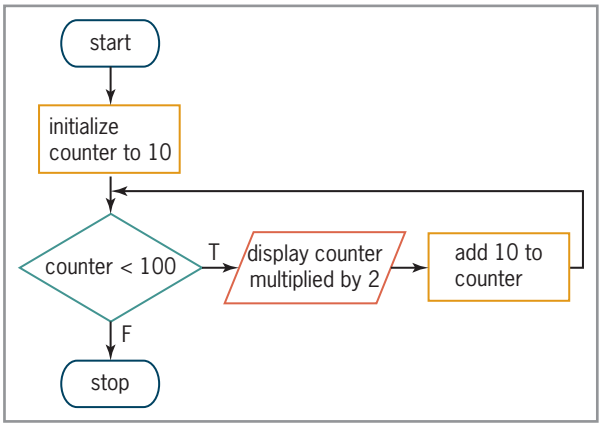


Figure 7-57

INTRODUCTORY

24. Figure 7-58 shows the flowchart for an algorithm that displays the numbers 10, 30, 50, 70, and 90 on the screen. Code the algorithm into a program; use the `for` statement. The counter variable should be an `int` variable named `number`. Then enter your C++ instructions into a source file named Introductory24.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Save and then run the program.

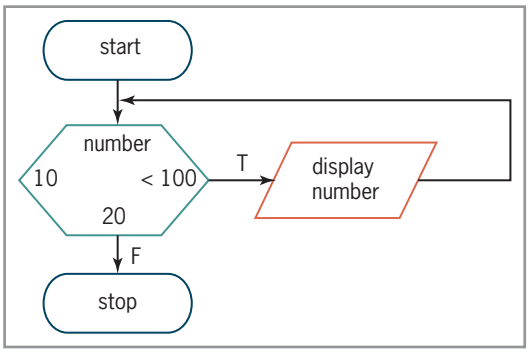


Figure 7-58

25. In this exercise, you create a program that displays the numbers 0 through 117, in increments of 9. Write two versions of the appropriate code: one using the `for` statement and the other using the `while` statement. Enter the C++ instructions into a source file named Introductory25.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Save and then run the program. The numbers 0, 9, 18 and so on should appear twice on the screen. INTRODUCTORY

26. The problem specification, IPO chart information, and C++ instructions for the Jasper Music Company program are shown in Figure 7-22 in the chapter. INTERMEDIATE

    a. Enter the C++ instructions into a source file named Intermediate26.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

    b. Save and then run the program. Test the program using the following sales amounts: 2500, 6000, and 2000.

    c. Modify the program so it uses the `for` statement rather than the `while` statement.

    d. Save and then run the program. Test the program using the following sales amounts: 2500, 6000, and 2000.

27. The problem specification, IPO chart information, and C++ instructions for the Holmes Supply Company program are shown in Figure 7-30 in the chapter. INTERMEDIATE

    a. Enter the C++ instructions into a source file named Intermediate27.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

    b. Save and then run the program. Test the program using the following payroll amounts: 15000, 25000, and 60000.

    c. Modify the program so it uses the `while` statement rather than the `for` statement.

    d. Save and then run the program. Test the program using the following payroll amounts: 15000, 25000, and 60000.

INTERMEDIATE

28. The problem specification, IPO chart information, and C++ instructions for the Colfax Sales program are shown in Figure 7-36 in the chapter.

    a. Enter the C++ instructions into a source file named Intermediate28.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

    b. Save and then run the program. Test the program using the number 25000 as the sales amount.

    c. Modify the program so it uses the `while` statement rather than the `for` statement.

    d. Save and then run the program. Test the program using the number 25000 as the sales amount.

INTERMEDIATE

29. Effective January 1st of each year, Gabriela receives a 5% raise on her previous year's salary. She wants a program that calculates and displays the amount of her annual raises for the next three years. The program also should calculate and display her total salary for the three years.

    a. Create an IPO chart for the problem, and then desk-check the algorithm using an annual salary of $10,000. (The raise amounts are $500.00, $525.00, and $551.25. The total salary is $33,101.25.)

    b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 7-13. Then code the algorithm into a program.

    c. Desk-check the program using the same data used to desk-check the algorithm.

    d. Enter your C++ instructions into a source file named Intermediate29.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

    e. Save and then run the program. Test the program using the same data used to desk-check the program.

ADVANCED

30. In this exercise, you create a program that displays the sum of the even integers between and including two numbers entered by the user. In other words, if the user enters an even number, that number should be included in the sum. For example, if the user enters the integers 2 and 7, the sum is 12 (2 + 4 + 6). If the user enters the integers 2 and 8, the sum is 20 (2 + 4 + 6 + 8). Display an error message if the first integer entered by the user is greater than the second integer.

    a. Create an IPO chart for the problem, and then desk-check the algorithm three times. For the first desk-check, use the integers 1 and 5. For the second desk-check, use the integers 12 and 21. For the third desk-check, use the integers 50 and 3.

    b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 7-13. Then code the algorithm into a program.

c. Desk-check the program using the same data used to desk-check the algorithm.

d. Enter your C++ instructions into a source file named Advanced30.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

e. Save and then run the program. Test the program using the same data used to desk-check the program.

31. ADVANCED

The sales manager at Premium Paper wants a program that allows her to enter the company's income and expense amounts, which always will be integers. The number of income and expense amounts may vary each time the program is run. For example, the sales manager may need to enter five income amounts and three expense amounts. Or, she may need to enter 20 income amounts and 30 expense amounts. The program should calculate and display the company's total income, total expenses, and profit (or loss).

a. Create an IPO chart for the problem, and then desk-check the algorithm three times. For the first desk-check, use the following income amounts: 5000, 7500, and 3350. Also use the following expense amounts: 125 and 999. For the second desk-check, use the following income amounts: 2450, 6700, 9000, and 5600. Also use the following expense amount: 4000. For the third desk-check, use an income amount of 8000 and an expense amount of 10000.

b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 7-13. Then code the algorithm into a program.

c. Desk-check the program using the same data used to desk-check the algorithm.

d. Enter your C++ instructions into a source file named Advanced31.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

e. Save and then run the program. Test the program using the same data used to desk-check the program.

32. ADVANCED

In this exercise, you create a program for the sales manager at Computer Haven, a small business that offers motivational seminars to local companies. Figure 7-59 shows the charge for attending a seminar. Notice that the charge per person depends on the number of people the company registers. For example, the cost for four registrants is $400; the cost for two registrants is $300. The program should allow the sales manager to enter the number of registrants for as many companies as needed. When the sales manager has finished entering the data, the program should calculate and display the total number of people registered, the total charge for those registrants, and the average charge per registrant. For example, if one company registers four people and another company registers two people, the total number of people registered is six, the total charge is $700, and the average charge per registrant is $116.67.

Number of people a company registers	Charge per person ($)
1 – 3	150
4 – 9	100
10 or more	90

Figure 7-59

a. Create an IPO chart for the problem, and then desk-check the algorithm appropriately.

b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 7-13. Then code the algorithm into a program.

c. Desk-check the program using the same data used to desk-check the algorithm.

d. Enter your C++ instructions into a source file named Advanced32.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Display the average charge in fixed-point notation with two decimal places.

e. Save and then run the program. Test the program using the same data used to desk-check the program.

ADVANCED

33. In this exercise, you create a program that displays the first 10 Fibonacci numbers (1, 1, 2, 3, 5, 8, 13, 21, 34, and 55). Notice that, beginning with the third number in the series, each Fibonacci number is the sum of the prior two numbers. In other words, 2 is the sum of 1 plus 1, 3 is the sum of 1 plus 2, 5 is the sum of 2 plus 3, and so on. Write two versions of the code: one using the `while` statement and the other using the `for` statement. Enter the C++ instructions into a source file named Advanced33.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Save and then run the program. The Fibonacci numbers should appear twice on the screen.

SWAT THE BUGS

34. Follow the instructions for starting C++ and opening the SwatTheBugs34.cpp file. The file is contained in either the Cpp6\Chap07\SwatTheBugs34 Project folder or the Cpp6\Chap07 folder. Debug the program.

Answers to TRY THIS Exercises

Pencil and Paper

1. See Figure 7-60. The code will display the numbers 0, 1, 2, 3, and 4 on separate lines on the computer screen. The loop stops when the `temp` variable's value is 5.



Figure 7-60

2. See Figure 7-61. The code will display Number: 0, Number: 2, and Number: 4 on separate lines on the computer screen. The loop stops when the `num` variable's value is 6.



Figure 7-61

Computer

17. No answer required.

18. To complete the program, enter the following `while` clause: `while (number >= 0)`.

19. To complete the program, enter the following `while` clause: `while (toupper(more) != 'N')`.

CHAPTER 8

More on the Repetition Structure

After studying Chapter 8, you should be able to:

- Include a posttest loop in pseudocode
- Include a posttest loop in a flowchart
- Code a posttest loop using the C++ `do while` statement
- Nest repetition structures
- Raise a number to a power using the `pow` function

Posttest Loops

Recall that a repetition structure can be either a pretest loop or a posttest loop. The difference between both types of loops pertains to when the loop's condition is evaluated. Unlike a pretest loop's condition, which is evaluated *before* the instructions within the loop are processed, a posttest loop's condition is evaluated *after* the instructions within the loop are processed. As a result, the instructions in a posttest loop will always be processed at least once, whereas the instructions in a pretest loop may never be processed. You learned about pretest loops in Chapter 7. You will learn about posttest loops in this chapter. Although pretest loops are the most commonly used, it is essential to understand the way posttest loops work. You may encounter a situation where a posttest loop is the better choice. Or, you may encounter a posttest loop in another programmer's code that you are either modifying or debugging.



Pretest and posttest loops also are called top-driven and bottom-driven loops, respectively.

The problem specification and algorithms shown in Figure 8-1 will help clarify the difference between pretest and posttest loops. Algorithm 1 contains a pretest loop, and Algorithm 2 contains a posttest loop. The purpose of the loop in each algorithm is to position Robin directly in front of her bedroom door. Compare the first and last lines in the pretest loop with the first and last lines in the posttest loop. More specifically, notice the location of the loop's condition. In the pretest loop, the condition appears in the first line, which indicates that Robin should evaluate it *before* she follows the instructions in the loop. In the posttest loop, the condition appears in the last line, indicating that Robin should evaluate it only *after* following the instructions in the loop. The pretest loop in Algorithm 1 will work when Robin is zero or more steps away from her bedroom door. The posttest loop in Algorithm 2, however, will work only when Robin is at least one step away from the door.

Robin is standing in her hallway facing her bedroom door. The door, which may or may not be closed, is an unknown number of steps away from her. Robin wants to go inside her bedroom.

Algorithm 1 – pretest loop

condition

```
1. repeat while (you are not directly in front of the bedroom door)
       walk forward
   end repeat
2. if (the bedroom door is closed)
       open the bedroom door
   end if
3. walk forward
```

works when Robin is zero or more steps away from the door

Algorithm 2 – posttest loop

```
1. repeat
       walk forward
   end repeat while (you are not directly in front of the bedroom door)
2. if (the bedroom door is closed)
       open the bedroom door
   end if
3. walk forward
```

condition

works only when Robin is at least one step away from the door

Figure 8-1 Problem specification and algorithms containing pretest and posttest loops

To understand why the loops in Figure 8-1 are not interchangeable, you will desk-check them twice. For the first desk-check, Robin is one step away from her bedroom door. In the pretest loop, the loop's condition checks Robin's current location. Robin is not directly in front of the door, so she is told to walk forward and then the loop's condition is evaluated again. Robin is now positioned correctly in front of her bedroom door, so the loop ends and Robin continues to the second instruction in the algorithm. The posttest loop, on the other hand, instructs Robin to walk forward, which places her directly in front of her bedroom door. The loop's condition is evaluated next. The condition checks whether Robin is positioned correctly; she is, so the loop ends and Robin continues to the second instruction in the algorithm. Notice that when Robin is one step away from the door, the pretest and posttest loops produce the same result: both place her right in front of the door. For the second desk-check, Robin is standing directly in front of her bedroom door. The condition in the pretest loop checks Robin's current location. Robin is already positioned correctly, so the walk forward instruction is bypassed and the loop ends. The posttest loop, on the other hand, instructs Robin to walk forward before the loop's condition is evaluated. But if Robin walks forward, she will bump into the door. Obviously, the posttest loop in Algorithm 2 does not work correctly when Robin starts out directly in front of her bedroom door. You can fix this problem by adding a selection structure to the algorithm. The modified algorithm is shown in Figure 8-2.

works when Robin is zero or more steps away from the door

Modified Algorithm 2 – posttest loop

```
1. if (you are not directly in front of the bedroom door)
        repeat
            walk forward
        end repeat while (you are not directly in front of the bedroom door)
   end if
2. if (the bedroom door is closed)
        open the bedroom door
   end if
3. walk forward
```

Figure 8-2 Selection structure added to Algorithm 2 from Figure 8-1

The posttest loop in Figure 8-2 is identical to the posttest loop in Figure 8-1, except it is processed only when the selection structure's condition evaluates to true, which is when Robin is not directly in front of her bedroom door. To understand how the selection and repetition structures in Figure 8-2 work, you will desk-check that portion of the algorithm twice. For the first desk-check, Robin is one step away from her bedroom door. The algorithm in Figure 8-2 begins with a single-alternative selection structure whose condition checks Robin's initial location. The condition evaluates to true because Robin is not directly in front of her bedroom door. Therefore, the instructions in the selection structure's true path are processed. The first instruction in the true path (repeat) marks the beginning of a posttest loop. Next, the instruction in the loop body directs Robin to walk forward. The loop's condition is evaluated next. At this point, Robin is positioned correctly; so the loop ends and so does the selection structure. Robin now continues to the second instruction in the algorithm. For the second desk-check, Robin is directly in front of her bedroom door. Here again, the condition in the selection

structure checks Robin's initial location. In this case, Robin is already in front of her bedroom door, so the selection structure's condition evaluates to false. As a result, the selection structure ends without processing the posttest loop contained in its true path. Robin simply continues to the second instruction in the algorithm. Although the modified algorithm works correctly, most programmers prefer to use a pretest loop, rather than a posttest loop with a selection structure, because it is easier to write and understand. Posttest loops should be used only when their instructions must be processed at least once. You often will find a posttest loop in programs that allow the user to select from a menu, such as a game program. This type of program uses the posttest loop to control the display of the menu, which must appear on the screen at least once.

Flowcharting a Posttest Loop

For many people, it's easier to understand the difference between a pretest loop and a posttest loop by viewing both loops in flowchart form. Figure 8-3 shows the problem specification for the Miller Incorporated program from Chapter 7. It also shows two correct algorithms in flowchart form. Algorithm 1, which you viewed in Figure 7-6 in Chapter 7, uses a pretest loop to get the sales amounts from the user; Algorithm 2 uses a posttest loop. Notice that the decision diamond, which contains the loop's condition, appears at the top of a pretest loop; however, it appears at the bottom of a posttest loop.

Problem specification
In January of each year, Miller Incorporated pays a 5% bonus to each of its salespeople. The bonus is based on the amount of sales made by the salesperson during the previous year. The payroll clerk wants a program that calculates and displays the bonus amounts for as many salespeople as needed without having to run the program more than once. Because the sales amounts entered by the payroll clerk will always be positive numbers, the payroll clerk will indicate that he is finished with the program by entering a sales amount of –1 (a negative number 1).

Input
bonus rate (5%)
sales

Processing
Processing items: none

Output
bonus

Algorithm 1 (pretest loop):

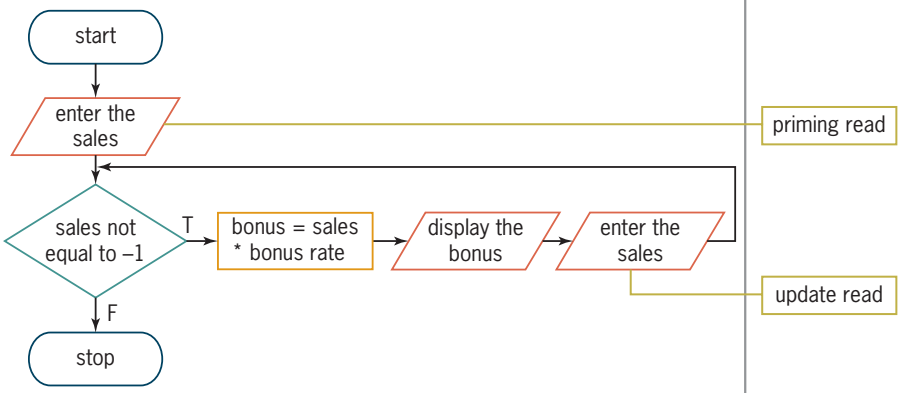


Figure 8-3 Miller Incorporated problem specification along with two algorithms shown in flowchart form (*continues*)

(*continued*)

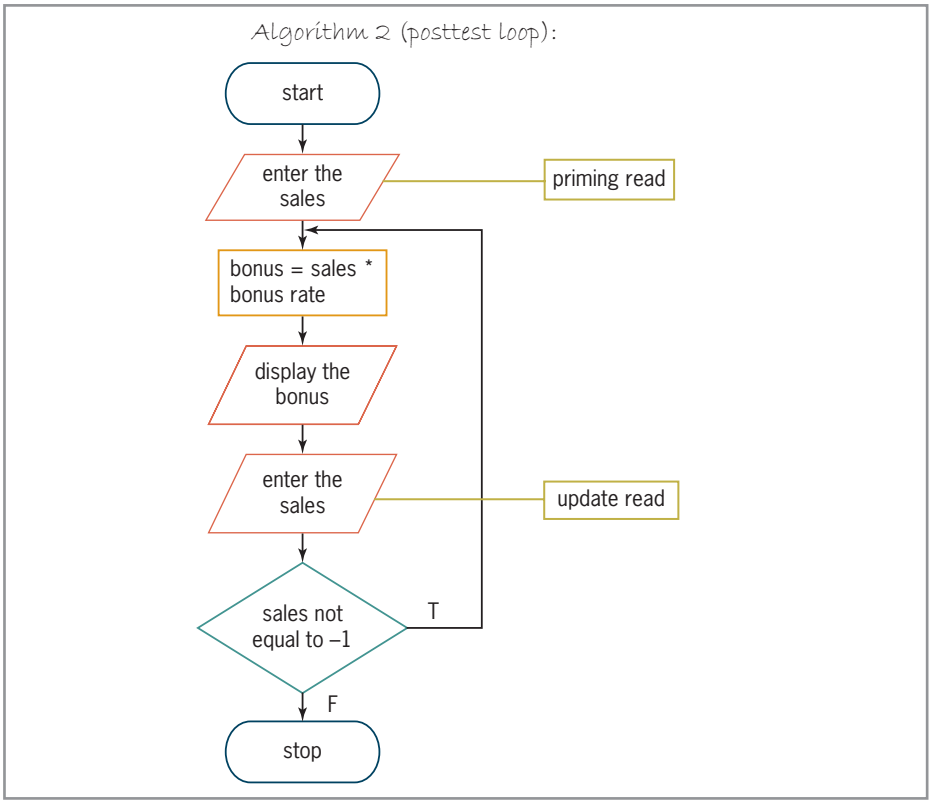


Figure 8-3 Miller Incorporated problem specification along with two algorithms shown in flowchart form



Recall that the oval, rectangle, parallelogram, and diamond in a flowchart are called the start/stop, processing, input/output, and decision symbols.

You can observe how a posttest loop operates in a program by desk-checking Algorithm 2 in Figure 8-3. You will desk-check it using sales amounts of 10000 and 25000, followed by the sentinel value (-1). The first input/output symbol in the flowchart represents the priming read. In this case, the priming read gets the first sales amount (10000) from the user. The instruction in the processing symbol is handled next and is the first instruction in the loop body. The instruction calculates the bonus by multiplying the sales amount by the bonus rate. Notice that the first sales amount is not compared to the sentinel value in a posttest loop. Figure 8-4 shows the first sales and bonus amounts recorded in the desk-check table.

bonus rate	sales	bonus
.05	10000	500

Figure 8-4 First sales and bonus amounts recorded in the desk-check table

The first input/output symbol in the loop body contains the instruction to display the bonus on the screen. In this case, the number 10000 will be displayed. The second input/output symbol in the loop body gets the next salesperson's sales (25000) from the user and represents the update read. Figure 8-5 shows the second sales amount recorded in the desk-check table.

bonus rate	sales	bonus
.05	~~10000~~	500
	25000	

Figure 8-5 Second sales amount recorded in the desk-check table

The next symbol in the flowchart is the decision diamond. The diamond contains the posttest loop's condition, which always marks the end of a posttest loop. The condition, which is phrased as a looping condition, compares the current sales amount with the sentinel value to determine whether the loop should be processed again (a true condition) or end (a false condition). Notice that this is the first time the loop's condition is evaluated. In this case, the condition evaluates to true because 25000 is not equal to -1. When the condition evaluates to true, the computer processes the instructions in the loop body. Those instructions calculate and display the bonus for the second salesperson and then get another sales amount (in this case, the sentinel value) from the user. Figure 8-8 shows the current status of the desk-check table.

bonus rate	sales	bonus
.05	~~10000~~	~~500~~
	~~25000~~	1250
	-1 sentinel value	

Figure 8-6 Current status of the desk-check table

Next, the loop's condition is reevaluated to determine whether the loop should be processed again (a true condition) or end (a false condition). In this case, the condition evaluates to false because the current sales amount is equal to the sentinel value. When the condition evaluates to false, the loop ends and processing continues with the instruction immediately following the loop. In Algorithm 2's flowchart in Figure 8-3, the loop is followed by the stop oval, which marks the end of the flowchart.

Mini-Quiz 8-1



The answers to Mini-Quiz questions are located in Appendix A.

1. If the user enters the numbers 5, 8, 9, and -1, how many times will the condition in the following algorithm be evaluated?

```
1. enter number
2. repeat while (number is greater than or equal to 0)
      display number
      enter number
   end repeat
```

2. If the user enters the number -4, how many times will the condition in Question 1's algorithm be evaluated?

3. If the user enters the numbers 5, 8, 9, and -1, how many times will the condition in the following algorithm be evaluated?

```
1. enter number
2. repeat
      display number
      enter number
   end repeat while (number is greater than or equal to 0)
```

4. If the user enters the numbers 0 and -4, how many times will the condition in Question 3's algorithm be evaluated?

The do while Statement



Recall that a looping condition specifies the requirement for processing the loop body instructions.

C++ provides the `do while` statement for coding a posttest loop. The statement's syntax is shown in Figure 8-7. As the boldfaced text in the syntax indicates, essential components of the statement include the `do` and `while` keywords, the parentheses that surround the condition, the braces, and the semicolon. The italicized items indicate where the programmer must supply information. In this case, the programmer must supply the loop's condition, which must be phrased as a looping condition. As in the `while` statement, the condition in the `do while` statement must evaluate to either true or false. The condition can contain variables, constants, functions, arithmetic operators, comparison operators, and logical operators. Besides providing the condition, the programmer also must provide the statements to be processed when the condition evaluates to true. Although not a requirement, some programmers use a comment (such as `//begin loop`) to mark the beginning of the `do while` statement, because it makes the program easier to read and understand. Also included in Figure 8-7 are examples of using the `do while` statement. The `while` clause in Example 1 tells the computer to repeat the loop body instructions as long as (or while) the value in the `age` variable is greater than zero. The loop will stop when the second number entered by the user is either the number 0 or a negative number. The `while` clause in Example 2 indicates that the loop body instructions should be repeated as long as the value in the `makeEntry` variable is either the uppercase letter Y or the lowercase letter y. In this case, the loop will stop when the user's second entry is anything other than the letter Y entered in either uppercase or lowercase. You also could write the last line in the loop as `} while (toupper(makeEntry) == 'Y');` or `} while (tolower(makeEntry) == 'y');`.

HOW TO Use the do while Statement

Syntax

```
do   //begin loop
{
     one or more statements to be processed one time, and thereafter
     as long as the condition is true
}    while (condition);
```

the statement ends with a semicolon

Example 1

```
int age = 0;

cout << "Enter an age greater than 0: ";
cin >> age;
do   //begin loop
{
     cout << "You entered " << age << endl << endl;
     cout << "Enter an age greater than 0: ";
     cin >> age;
}    while (age > 0);
```

priming read

update read

semicolon

Example 2

```
char makeEntry = ' ';
double sales = 0.0;

cout << "Enter a sales amount? (Y/N) ";
cin >> makeEntry;
do   //begin loop
{
     cout << "Enter the sales: ";
     cin >> sales;
     cout << "You entered " << sales << endl << endl;
     cout << "Enter a sales amount? (Y/N) ";
     cin >> makeEntry;
}    while (makeEntry == 'Y' || makeEntry == 'y');
```

priming read

update read

semicolon

Figure 8-7 How to use the do while statement



The braces in a do while statement are not required when the loop body contains only one statement. However, a one-statement loop body is rare.

Earlier, in Figure 8-3, you viewed the problem specification and IPO chart for the Miller Incorporated program. The figure showed two algorithms in flowchart form. Algorithm 1 used a pretest loop to solve the problem, whereas Algorithm 2 used a posttest loop. Figure 8-8 shows the pseudocode and C++ instructions corresponding to Algorithm 2. The first three statements in the figure declare and initialize the RATE named constant and the sales and bonus variables. Next, the cout statement prompts the user to enter the first sales amount, and the cin statement stores the user's response in the sales variable. The do clause appears next in the code and marks the beginning of a posttest loop. The first three instructions in the loop body calculate the bonus amount and then display the result, followed by two blank lines, on the computer screen. The last two instructions in the

loop body prompt the user to enter the next sales amount and then store the user's response in the `sales` variable. Notice that the loop body instructions are processed *before* the condition in the `while` clause is evaluated. After the loop body instructions are processed, the condition in the `while` clause compares the value stored in the `sales` variable with the sentinel value. If the `sales` variable does not contain the sentinel value, the condition evaluates to true and the loop body instructions are processed again. Those instructions calculate the bonus, then display the bonus and two blank lines, then prompt the user to enter the next sales amount, and then store the user's response in the `sales` variable. From now on, each time the user enters a sales amount, the condition in the `while` clause compares the sales amount with the sentinel value. When the user enters the sentinel value—in this case, -1—as the sales amount, the loop body instructions are not processed again. Instead, the loop is exited and processing continues with the instruction located immediately below the loop. A sample run of the Miller Incorporated program is shown in Figure 8-9. (The program uses the `fixed` and `setprecision` stream manipulators to display the bonus amounts in fixed-point notation with two decimal places.)



The `while` clause in Figure 8-8 contains a looping condition.

IPO chart information	**C++ instructions**
Input	
bonus rate (5%)	`const double RATE = .05;`
sales	`int sales = 0;`
Processing	
none	
Output	
bonus	`double bonus = 0.0;`
Algorithm	
1. enter the sales	`cout << "First sales amount (-1 to stop): ";`
priming read	`cin >> sales;`
2. repeat	`do //begin loop`
	`{`
calculate the bonus by multiplying the sales by the bonus rate	`bonus = sales * RATE;`
display the bonus	`cout << "Bonus: $" << bonus;`
	`cout << endl << endl;`
enter the sales	`cout << "Next sales amount (-1 to stop): ";`
update read	`cin >> sales;`
end repeat while (the sales are not equal to -1)	`} while (sales != -1);`

Figure 8-8 IPO chart information and C++ instructions for the Miller Incorporated program

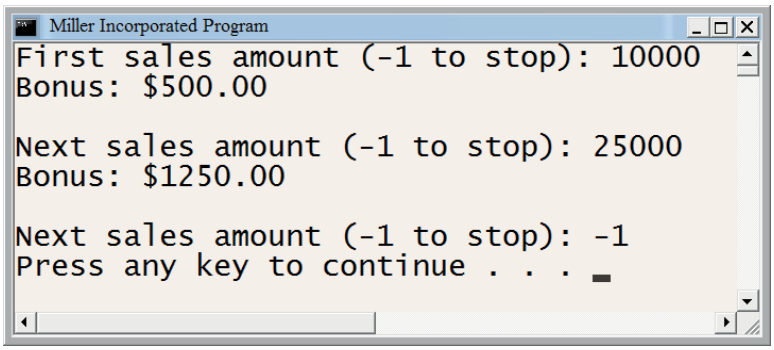

Miller Incorporated Program
First sales amount (-1 to stop): 10000
Bonus: $500.00
Next sales amount (-1 to stop): 25000
Bonus: $1250.00
Next sales amount (-1 to stop): -1
Press any key to continue . . .

Figure 8-9 A sample run of the Miller Incorporated program

Mini-Quiz 8-2



The answers to Mini-Quiz questions are located in Appendix A.

1. The `do while` clause marks the beginning of the C++ `do while` statement.
   a. True
   b. False

2. The `while` clause in the C++ `do while` statement ends with a _____.
   a. brace
   b. colon
   c. comma
   d. semicolon

3. Write a C++ `while` clause that processes a posttest loop's instructions as long as the value in the `inStock` variable is greater than the value in the `reorder` variable.

4. Write a C++ `while` clause that processes a posttest loop's instructions as long as the value in a `char` variable named `letter` is either Y or y. Use the built-in `toupper` function.

Nested Repetition Structures

Like selection structures, repetition structures can be nested. In other words, you can place one loop (called the nested or inner loop) within another loop (called the outer loop). Both loops can be pretest loops, or both can be posttest loops. Or, one can be a pretest loop and the other a posttest loop. A programmer determines whether a problem's solution requires a **nested loop** by studying the problem specification. Figure 8-10 shows a problem specification and algorithm from Chapter 7. The algorithm requires a loop

because the instructions for signing a book need to be repeated for every customer. However, the algorithm does not require a nested loop. This is because all of the instructions within the loop should be followed only once per customer.

Robin is sitting at a table in a bookstore, attending her book signing. Customers are standing in line waiting for Robin to sign their copy of her bestselling book on Robotics. Robin needs to sign each customer's book.

```
repeat while (there are customers in line)
      accept the book from the customer
      place the book on the table
      open the front cover of the book
      sign your name on the first page
      close the book
      return the book to the customer
      thank the customer
end repeat
```

follow these instructions for each customer

Figure 8-10 Problem specification and algorithm for signing one book for each customer

Now consider the possibility that a customer may have more than one book for Robin to sign. It's also possible that a customer, not knowing about the book signing in advance, has left his book at home and is standing in line for the sole purpose of meeting Robin. What changes will need to be made to the algorithm in Figure 8-10? Instead of being repeated once for each customer in line, the first six instructions in the loop body—the instructions that begin with *accept the book from the customer* and end with *return the book to the customer*—now must be repeated for an unknown number of books for each of an unknown number of customers. Therefore, in addition to requiring the current loop, which determines whether there is a customer in line, the modified algorithm will require a nested loop to determine whether the customer has any books that need signing. Robin will thank the customer only after all of the customer's books have been signed, so the last instruction in the loop (the *thank the customer* instruction) needs to be repeated only once for each customer. It does not need to be repeated for each book. Therefore, it should be included in the outer loop, but not in the nested loop. Figure 8-11 shows the modified problem specification and algorithm. The outer loop begins with *repeat while (there are customers in line)*, and it ends with the last *end repeat*. The nested loop begins with *repeat while (the customer has a book that needs signing)*, and it ends with the first *end repeat*. The outer loop's instructions will be followed for each customer in line; the outer loop ends when there are no more customers. The nested loop's instructions will be followed for each book the current customer wants signed. The nested loop ends when the customer has no more books to sign. Notice that the entire nested loop is contained within the outer loop. This must be true for the loop to be nested and work correctly.

Robin is sitting at a table in a bookstore, attending her book signing. Customers are standing in line waiting for Robin to sign their copy of her bestselling book on Robotics. Robin needs to sign each customer's book. It's possible that some customers may not have a book, while others may have more than one book.

```
repeat while (there are customers in line)
      repeat while (the customer has a book that needs signing)
            accept the book from the customer
            place the book on the table
            open the front cover of the book
            sign your name on the first page
            close the book
            return the book to the customer
      end repeat
      thank the customer
end repeat
```

follow these instructions for each book

follow these instructions for each customer

Figure 8-11 Modified problem specification and algorithm for signing zero or more books for each customer

A clock uses nested repetition structures to keep track of the time. For simplicity, consider a clock's minute and second hands only. The second hand on a clock moves one position, clockwise, for every second that has elapsed. After the second hand moves 60 positions, the minute hand moves one position, also clockwise. The second hand then begins its journey around the clock again. Figure 8-12 shows the logic used by a clock's minute and second hands. As the figure indicates, an outer loop controls the minute hand, while an inner (nested) loop controls the second hand. Here again, notice that the entire nested loop is contained within the outer loop. Recall that this is a requirement for the loop to be nested and work correctly.

```
1. start minutes at 0
2. repeat while (minutes are less than 60)
      start seconds at 0
      repeat while (seconds are less than 60)
            move second hand 1 position, clockwise
            add 1 to seconds
      end repeat
      move minute hand 1 position, clockwise
      add 1 to minutes
   end repeat
```



The next iteration of the outer loop (which controls the minute hand) occurs only after the nested loop (which controls the second hand) has finished processing.

Figure 8-12 Logic used by a clock's minute and second hands

The Asterisks Program

Figure 8-13 shows the problem specification, IPO chart information, and C++ instructions for a program that displays an asterisk on three separate lines on the computer screen. The algorithm, which is shown in both pseudocode and a flowchart, does not require a nested loop. This is because

all of the instructions within the loop need to be repeated the same number of times: three. Figure 8-14 shows the completed desk-check table, and Figure 8-15 shows a sample run of the program.

Problem specification
Create a program that displays an asterisk on three separate lines on the computer screen, like this:
*
*
*

IPO chart information	**C++ instructions**
Input	
none	
Processing	
number of lines (counter: 1 to 3)	this variable is created and initialized in the for clause
Output	
asterisk (on each of 3 lines)	
Algorithm	
repeat while (number of lines is less than 4)	`for (int line = 1; line < 4; line += 1)`
	`{`
display an asterisk	`cout << '*';`
position the cursor on the next line	`cout << endl;`
end repeat	`} //end for`

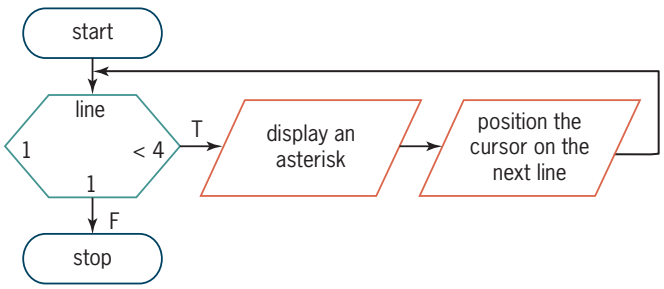


Figure 8-13 Problem specification, IPO chart information, and C++ instructions for the asterisks program

line
1
2
3
4

Figure 8-14 Completed desk-check table for the asterisks program

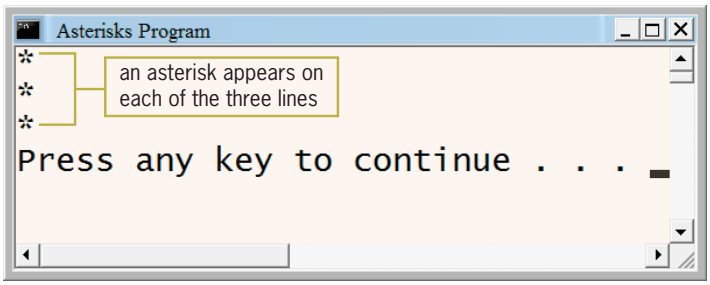


Figure 8-15 Sample run of the asterisks program

Figure 8-16 shows a modified version of the previous problem specification, along with the corresponding IPO chart information and C++ instructions. In the modified version, the program needs to display five asterisks on each of three separate lines on the computer screen. The modified algorithm will use two loops to accomplish this task: an outer pretest loop to keep track of the number of lines and a nested pretest loop to keep track of the number of asterisks. The changes made to the IPO chart information and C++ instructions are shaded in Figure 8-16. In the code, the outer loop begins with the first `for` clause, which directs the computer to repeat the loop body instructions three times. The outer loop ends with the `} //end for` line. Braces are required in the outer loop because its loop body contains more than one statement. The nested loop begins with the second `for` clause, which directs the computer to repeat the `cout << '*';` statement five times. The nested loop ends with the `//end for` comment. Braces are not needed in the nested loop because its loop body contains only one statement. Here again, notice that the entire nested loop is contained within the outer loop. Although both loops in Figure 8-16 are coded using the `for` statement, one or both could be coded using the `while` statement. In addition, the algorithm could have been written using one or more posttest loops. Recall that you use the `do while` statement to code a posttest loop in C++.

Problem specification
Create a program that displays five asterisks on each of three separate lines on the computer screen, like this:

```
*****
*****
*****
```

IPO chart information	**C++ instructions**
Input	
none	
Processing	
number of lines (counter: 1 to 3)	this variable is created and initialized in the for clause
number of asterisks (counter: 1 to 5)	this variable is created and initialized in the for clause
Output	
asterisk (5 on each of 3 lines)	

Figure 8-16 Problem specification, IPO chart information, and C++ instructions for the modified asterisks program (*continues*)

(*continued*)

Algorithm

```
repeat while (number of
lines is less than 4)
    repeat while (number
    of asterisks is less than 6)
        display an asterisk
    end repeat
    position the cursor on the next line
end repeat
```

```
for (int line = 1;
line < 4; line += 1)
{
    for (int numAsterisks = 1;
    numAsterisks < 6;
    numAsterisks += 1)
        cout << '*';
    //end for
    cout << endl;
}   //end for
```

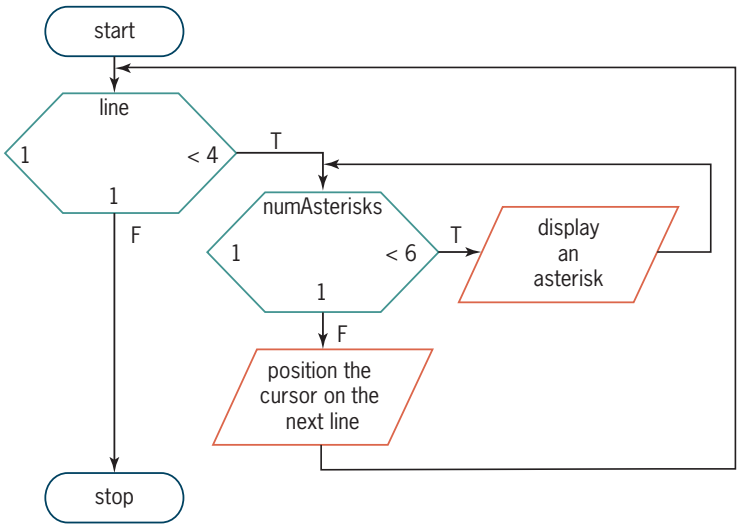


Figure 8-16 Problem specification, IPO chart information, and C++ instructions for the modified asterisks program

You can observe the way the computer processes a nested loop by desk-checking the code shown in Figure 8-16. First, the *initialization* argument in the outer loop's `for` clause (`int line = 1`) tells the computer to create the `line` variable and initialize it to 1. The *condition* argument (`line < 4`) then directs the computer to check whether the `line` variable's value is less than the number 4. It is, so the instructions in the body of the outer loop are processed. The first instruction is the nested loop's `for` clause. The clause's *initialization* argument (`int numAsterisks = 1`) tells the computer to create the `numAsterisks` variable and initialize it to the number 1. Its *condition* argument (`numAsterisks < 6`) then directs the computer to check whether the value in the `numAsterisks` variable is less than 6. It is, so the `cout << '*';` statement in the body of the nested loop is processed. That statement displays an asterisk on the current line on the computer screen. Figure 8-17 shows the desk-check table and output after the nested loop's `cout` statement is processed the first time.



Recall from Chapter 7 that the `for` clause's *condition* argument must be phrased as a looping condition, which means it must specify the requirement for processing the loop body instructions.

```
line            numAsterisks
1               1

Output
*
```

Figure 8-17 Desk-check table and output after the nested loop's `cout` statement is processed the first time

Next, the *update* argument in the nested loop's `for` clause (`numAsterisks += 1`) tells the computer to add the number 1 to the value stored in the `numAsterisks` variable; the result is 2. Once again, the *condition* argument in the nested loop's `for` clause directs the computer to check whether the variable's value is less than 6. It is, so the nested loop's `cout << '*';` statement displays another asterisk on the current line on the computer screen. Figure 8-18 shows the desk-check table and output after the `cout` statement is processed the second time.

```
line            numAsterisks
1               1
                2

Output
**
```

Figure 8-18 Desk-check table and output after the nested loop's `cout` statement is processed the second time

The computer again adds the number 1 to the value stored in the `numAsterisks` variable, giving 3. It then checks whether the variable's value is less than 6. It is, so the nested loop's `cout << '*';` statement displays another asterisk on the current line on the computer screen. Figure 8-19 shows the desk-check table and output after the `cout` statement is processed the third time.

```
line            numAsterisks
1               1
                2
                3

Output
***
```

Figure 8-19 Desk-check table and output after the nested loop's `cout` statement is processed the third time

The value stored in the `numAsterisks` variable is updated again; this time, the result is 4. The computer then checks whether the variable's value is less than 6. It is, so the `cout << '*';` statement in the nested loop displays another asterisk on the current line on the computer screen. Figure 8-20 shows the desk-check table and output after the `cout` statement is processed the fourth time.

line	numAsterisks
1	1
	2
	3
	4

Output
```
****
```

Figure 8-20 Desk-check table and output after the nested loop's `cout` statement is processed the fourth time

Here again, the computer adds the number 1 to the value stored in the `numAsterisks` variable, giving 5. The computer then checks whether the variable's value is less than 6. It is, so the nested loop's `cout << '*';` statement displays another asterisk on the current line on the computer screen. Figure 8-21 shows the desk-check table and output after the `cout` statement is processed the fifth time.

line	numAsterisks
1	1
	2
	3
	4
	5

Output
```
*****
```

Figure 8-21 Desk-check table and output after the nested loop's `cout` statement is processed the fifth time



Recall from Chapter 7 that the variable created in the `for` clause is local to the `for` statement and is removed from memory when the `for` loop ends.

The value stored in the `numAsterisks` variable is updated once again; this time, the result is 6. The computer then checks whether the variable's value is less than 6. It's not, so the nested loop ends and the `numAsterisks` variable is removed from the computer's internal memory. Processing continues with the first statement following the end of the nested loop. In this case, the `cout << endl;` statement positions the cursor on the next line on the computer screen. The `cout << endl;` statement is the last statement in the body of the outer loop. Therefore, after processing the statement, the computer returns to the outer loop's `for` clause to process its *update* and *condition* arguments. The *update* argument (`line += 1`) tells the computer to add the number 1 to the value stored in the `line` variable; the result is 2. The *condition* argument (`line < 4`) directs the computer to check whether the variable's value is less than 4. It is, so the instructions in the body of the outer loop are processed again. The first instruction is the nested loop's `for` clause, whose *initialization* and *condition* arguments tell the computer to create the `numAsterisks` variable and initialize it to the number 1, and then check whether its value is less than 6. At this point, the variable's value is less than 6, so the `cout << '*';` statement in the nested loop is processed. That statement displays an asterisk on the second line on the computer screen. Figure 8-22 shows the current status of the desk-check table and output.

line	numAsterisks
1	1
2	2
	3
	4
	5
	6
	1

Output

```
*****
*
```

Figure 8-22 Current status of the desk-check table and output



Notice that the nested loop is completely processed prior to the next iteration of the outer loop.

Next, the computer updates the value in the `numAsterisks` variable by adding the number 1 to it; the result is 2. The computer then checks whether the variable's value is less than 6. It is, so the nested loop's `cout << '*';` statement displays another asterisk on the current line on the computer screen. The computer again adds the number 1 to the value stored in the `numAsterisks` variable, giving 3. It then checks whether the variable's value is less than 6. It is, so the `cout << '*';` statement displays the third asterisk on the current line on the computer screen. Once again, the computer adds the number 1 to the value stored in the `numAsterisks` variable and then checks whether the value still is less than 6. At this point, the variable contains the number 4, so the `cout << '*';` statement displays the fourth asterisk on the current line on the computer screen. The computer again adds the number 1 to the value stored in the `numAsterisks` variable and then checks whether the value (which is now 5) is less than 6. It is, so the `cout << '*';` statement displays the fifth asterisk on the current line on the computer screen. Once again, the computer updates the value stored in the `numAsterisks` variable by adding the number 1 to it; the result is 6. The computer then checks whether the variable's value is less than 6. It's not, so the nested loop ends and the `numAsterisks` variable is removed from the computer's internal memory. Figure 8-23 shows the desk-check table and output after the nested loop ends the second time.

line	numAsterisks
1	1
2	2
	3
	4
	5
	6
	1
	2
	3
	4
	5
	6

Output

```
*****
*****
```

Figure 8-23 Desk-check table and output after the nested loop ends the second time

Processing continues with the first statement located below the nested loop—in this case, the `cout << endl;` statement. That statement positions the cursor on the next line on the computer screen. After processing the statement, the computer returns to the outer loop's `for` clause to process its *update* and *condition* arguments. The *update* argument (`line += 1`) tells the computer to add the number 1 to the value stored in the `line` variable; the result is 3. The *condition* argument (`line < 4`) directs the computer to check whether the variable's value is less than 4. It is, so the instructions in the body of the outer loop are processed again. The first instruction is the nested loop's `for` clause, whose *initialization* and *condition* arguments tell the computer to create the `numAsterisks` variable and initialize it to the number 1, and then check whether its value is less than 6. At this point, the variable's value is less than 6, so the `cout << '*';` statement in the nested loop is processed. That statement displays the first asterisk on the third line on the computer screen. The computer will continue processing the nested loop until the value stored in the `numAsterisks` variable is the number 6. Figure 8-24 shows the desk-check table and output after the nested loop ends the third time.

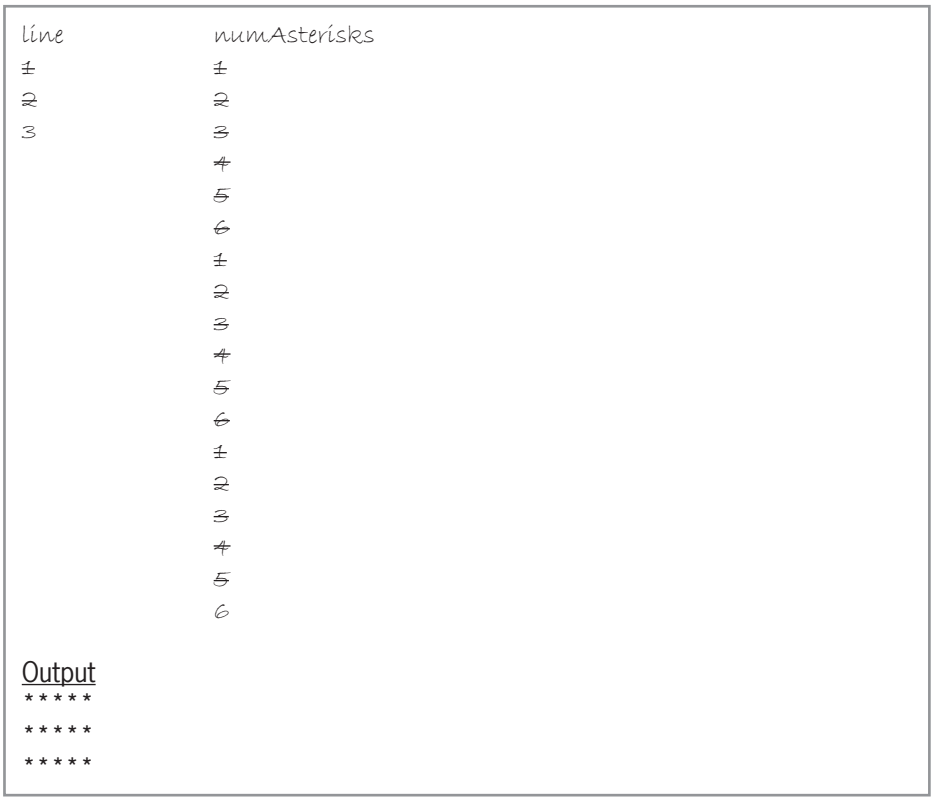


Figure 8-24 Desk-check table and output after the nested loop ends the third time

Processing continues with the `cout << endl;` statement, which follows the nested loop and is the last statement in the body of the outer loop. The statement positions the cursor on the next line on the computer screen. After processing the statement, the computer returns to the outer loop's `for` clause to process its *update* and *condition* arguments. The *update* argument (`line += 1`) tells the computer to add the number 1 to the value stored in the `line` variable; the result is 4. The *condition* argument (`line < 4`) directs the computer to check whether the variable's value is less than 4. It's not, so the outer loop ends and the `line` variable is removed from the computer's internal

memory. Processing will continue with the first statement located below the end of the outer loop. Figure 8-25 shows a sample run of the modified asterisks program.

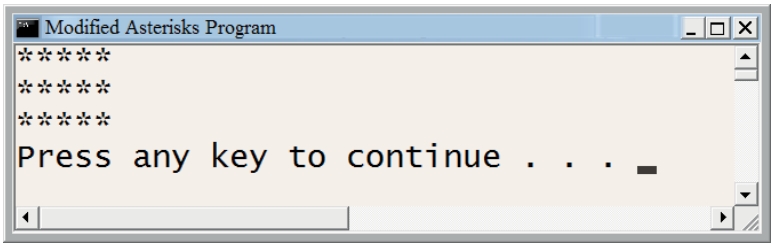


Figure 8-25 Sample run of the modified asterisks program

The Savings Calculator Program

Figure 8-26 shows the problem specification for the savings calculator program, which calculates the value of a one-time deposit into a savings account that earns a specific amount of interest for a certain amount of time. The figure also includes sample calculations. The formula in Example 1 calculates the savings account balance at the end of the first year. It does this by adding the number 1 to the interest rate of .03 and then raising the sum (1.03) to the first power. Recall that the result of raising a number to a power of 1 is the number itself; in this case, the result is 1.03. The formula in Example 1 then multiplies the number 1.03 by the number 1000, giving 1030. The formula in Example 2 calculates the account balance at the end of the third year. Here again, the formula begins by adding the number 1 to the interest rate of .03. However, this time it raises the sum (1.03) to the third power. Raising a number to a power of 3 means multiplying it by itself three times; in this case, the expression 1.03 * 1.03 * 1.03 yields 1.092727. The formula then multiplies the number 1.092727 by the number 1000, giving 1092.73 when rounded to two decimal places. Figure 8-26 also contains the program's IPO chart and a desk-check table. Notice that the algorithm uses a counter to keep track of the years, which are from 1 to 5. It also uses a posttest loop to calculate and display the account balance at the end of each year. (The flowchart for this program is contained in the Ch8Flowcharts.pdf file, which is located in the Cpp6\Chap08 folder.)



You also can use a pretest loop to calculate and display the savings account balance.

Problem specification
For your 21st birthday, your grandmother opens a savings account for you and deposits $1000 into the account. The savings account pays a 3% interest on the account balance. If you don't deposit any more money into the account, and you don't withdraw any money from the account, how much will your savings account be worth at the end of 1 through 5 years? Create a program that gives you the answers. You can calculate the answers using the following formula: $b = p * (1 + r)^n$. In the formula, p is the principal (the amount of the deposit), r is the annual interest rate, n is the number of years, and b is the balance in the savings account at the end of the *nth* year.

Figure 8-26 Problem specification, sample calculations, IPO chart, and desk-check table for the savings calculator program (*continues*)

(*continued*)

Example 1
$b = 1000 * (1 + .03)^1$
$b = \$1030$

Example 2
$b = 1000 * (1 + .03)^3$
$b = \$1092.73$ (rounded to two decimal places)

Input	Processing	Output
principal (1000) annual interest rate (3%) number of years (counter: 1 to 5)	Processing items: none	account balance (at end of each of the 5 years)

```
Algorithm:
repeat
      calculate the account
      balance = principal
      * (1 + annual interest rate) number of years

      display the current number
      of years and account balance

      add 1 to the number of years
end repeat while (number of years
is less than 6)
```

principal	annual interest rate	years	account balance
1000	.03	~~1~~	~~1030~~
		~~2~~	~~1060.9~~
		~~3~~	~~1092.73~~ (rounded to two decimal places)
		~~4~~	~~1125.51~~ (rounded to two decimal places)
		~~5~~	1159.27 (rounded to two decimal places)
		6	

Figure 8-26 Problem specification, sample calculations, IPO chart, and desk-check table for the savings calculator program

Before coding the algorithm shown in Figure 8-26, you will learn about the built-in C++ pow function. The function provides a convenient way to code a formula that raises a number to a power.

The pow Function

Some mathematical expressions require a number to be raised to a power. Raising a number to a power is referred to as **exponentiation**. An example of exponentiation is found in the πr^2 expression, where r is the radius of a circle. The expression calculates the circle's area by raising its radius to the second power and then multiplying the result by pi. C++ provides the built-in pow function for performing exponentiation. The **pow function** raises a number

to a power and then returns the result as a **double** number. Figure 8-27 shows the function's syntax, which contains two arguments: *x* and *y*. The *x* argument represents the number you want raised to power *y*. At least one of the two arguments must be a **double** number. To use the **pow** function in a program, the program must contain the `#include <cmath>` directive. Also included in Figure 8-27 are examples of C++ statements that contain the **pow** function. For clarity, the variable declaration statements are included in the examples. The `cube = pow(4.0, 3);` statement in Example 1 raises the **double** number 4.0 to the third power. In other words, it multiplies the **double** number 4.0 by itself three times (4.0 * 4.0 * 4.0). The statement then assigns the result, which is the **double** number 64.0, to the `cube` variable. The `cout << pow(100, .5);` statement in Example 2 raises the number 100 to the .5 power and is equivalent to finding the square root of the number. The statement displays the number 10 on the screen. The `area = 3.14 * pow(radius, 2.0);` statement in Example 3 raises the number contained in the `radius` variable (5.0) to the second power. It then multiplies the result by 3.14 and assigns the product (78.5) to the `area` variable. Now that you know how to use the **pow** function, you can code the savings calculator program.



Raising a number to the second power is the same as squaring the number.



In Chapter 9, you will learn about the `sqrt` function, whose purpose is to find the square root of a number.

HOW TO Use the pow Function

Syntax

pow(*x***,** *y***)** — requires the `#include <cmath>` directive

Example 1

```
double cube = 0.0;
cube = pow(4.0, 3);
```

The assignment statement assigns the number 64.0, which is 4.0 raised to the third power, to the `cube` variable.

Example 2

```
cout << pow(100, .5);
```

The statement displays the number 10, which is 100 raised to the .5 power. The `pow(100, .5)` expression is equivalent to finding the square root of the number 100.

Example 3

```
double area = 0.0;
double radius = 5.0;
area = 3.14 * pow(radius, 2.0);
```

The assignment statement raises the value stored in the `radius` variable to the second power; in other words, it squares the value. The result is 25.0. The assignment statement then multiplies the 25.0 by 3.14 and assigns the product (78.5) to the `area` variable.



As you learned in Chapter 5, an item within the parentheses following a function's name is called an argument.



You would need to use an assignment statement to save the result of the **pow** function in Example 2.

Figure 8-27 How to use the **pow** function

Coding the Savings Calculator Program

Figure 8-28 contains the information from the IPO chart shown earlier in Figure 8-26. It also contains the corresponding C++ instructions. The pow function is shaded in the figure.

IPO chart information	C++ instructions
Input	
principal (1000)	`int principal = 1000;`
annual interest rate (3%)	`double rate = .03;`
number of years (counter: 1 to 5)	`int years = 1;`
Processing	
none	
Output	
account balance (at end of each of the 5 years)	`double balance = 0.0;`
Algorithm	
repeat	`do //begin loop`
	`{`
calculate the account balance = principal * (1 + annual interest rate)^number of years	`balance = principal * pow(1 + rate, years);`
display the current number of years and account balance	`cout << "Year " << years << ":" << endl;` `cout << " $" << balance << endl;`
add 1 to the number of years	`years += 1;`
end repeat while (number of years is less than 6)	`} while (years < 6);`

Figure 8-28 IPO chart information and C++ instructions for the savings calculator program

Figure 8-29 shows the complete savings calculator program, and Figure 8-30 shows a sample run of the program. Notice that the account balances in Figure 8-30 agree with the amounts in the desk-check table shown earlier in Figure 8-26.

```
 1  //Savings Calculator.cpp - displays the balance
 2  //in a savings account at the end of 1 through 5 years
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  #include <iomanip>
 7  #include <cmath>
 8  using namespace std;
 9
10  int main()
11  {
12      int principal  = 1000;
13      double rate    = .03;
14      int years      = 1;   //counter
15      double balance = 0.0;
16
17      //display output with two decimal places
18      cout << fixed << setprecision(2);
19
20      do  //begin loop
21      {
22          balance = principal * pow(1 + rate, years);
23          cout << "Year " << years << ":" << endl;
24          cout << "    $" << balance << endl;
25          //update years counter
26          years += 1;
27      }   while (years < 6);
28
29      system("pause");
30      return 0;
31  }  //end of main function
```

your C++ development tool may not require this statement

Figure 8-29 Savings calculator program

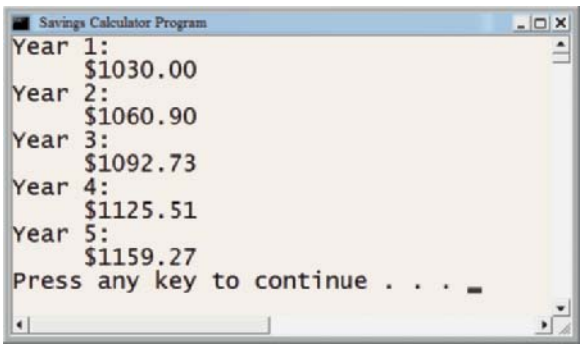


Figure 8-30 Sample run of the savings calculator program

Modifying the Savings Calculator Program

Currently, the savings calculator program calculates each account balance using an annual interest rate of 3%. In this section, you'll modify the program to use annual interest rates of 3%, 4%, and 5%. Figure 8-31 shows the modified IPO chart information and C++ instructions. The modifications made to

the original information and instructions, shown earlier in Figure 8-28, are shaded in Figure 8-31. Notice that the algorithm and C++ instructions now contain two loops: an outer posttest loop to keep track of the years, and a nested pretest loop to keep track of the rates. (The flowchart for the modified program is contained in the Ch8Flowcharts.pdf file, which is located in the Cpp6\Chap08 folder.)

IPO chart information	C++ instructions
Input	
principal (1000)	`int principal = 1000;`
annual interest rate (counter: 3% to 5%)	this variable is created and initialized in the nested for clause
number of years (counter: 1 to 5)	`int years = 1;`
Processing	
none	
Output	
account balance (at end of each of the 5 years)	`double balance = 0.0;`
Algorithm	
repeat	`do //begin loop`
	`{`
display the current number of years	`cout << "Year " << years << ":" << endl;`
repeat for (annual interest rate from 3% to 5%)	`for (double rate = .03; rate < .06; rate += .01)`
	`{`
calculate the account balance = principal * (1 + annual interest rate)$^{\text{number of years}}$	`balance = principal * pow(1 + rate, years);`
display the current rate with no decimal places	`cout << fixed << setprecision(0); cout << "Rate " << rate * 100 << "%: $";`
display the current account balance with two decimal places	`cout << setprecision(2) << balance << endl;`
end repeat	`} //end for`
add 1 to the number of years	`years += 1;`
end repeat while (years less than 6)	`} while (years < 6);`

Figure 8-31 Modified IPO chart information and C++ instructions

Figure 8-32 shows the modified savings calculator program. The nested loop is shaded in the figure. Figure 8-33 shows a sample run of the modified program.

```
 1  //Modified Savings Calculator.cpp - displays the balance
 2  //in a savings account at the end of 1 through 5 years
 3  //using interest rates of 3%, 4%, and 5%
 4  //Created/revised by <your name> on <current date>
 5
 6  #include <iostream>
 7  #include <iomanip>
 8  #include <cmath>
 9  using namespace std;
10
11  int main()
12  {
13      int principal  = 1000;
14      int years      = 1;      //counter
15      double balance = 0.0;
16
17      do  //begin loop
18      {
19          cout << "Year " << years << ":" << endl;
20
21          for (double rate = .03; rate < .06; rate += .01)
22          {
23              balance = principal * pow(1 + rate, years);
24              //display rate with zero decimal places
25              cout << fixed << setprecision(0);
26              cout << "   Rate " << rate * 100 << "%: $";
27              //display balance with two decimal places
28              cout << setprecision(2) << balance << endl;
29          }   //end for
30
31          //update years counter
32          years += 1;
33      }   while (years < 6);
34
35      system("pause");
36      return 0;
37  }   //end of main function
```

Figure 8-32 Modified savings calculator program

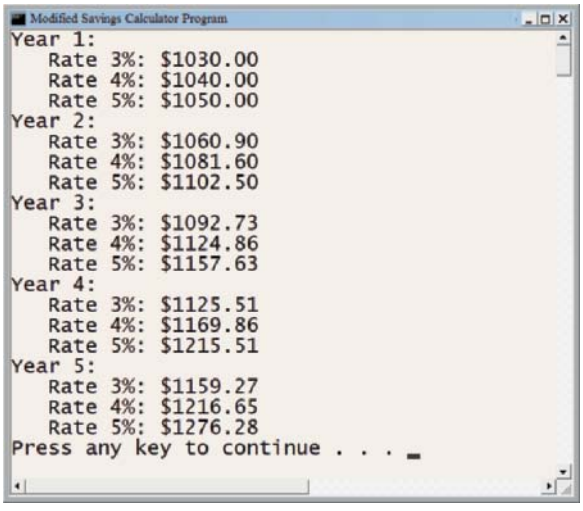


Figure 8-33 Sample run of the modified savings calculator program

The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 8-3

1. A nested loop can be a pretest loop only.
   a. True
   b. False

2. For a nested loop to work correctly, it must be contained entirely within an outer loop.
   a. True
   b. False

3. Consider a clock's hour and minute hands only. The hour hand is controlled by a(n) _____ loop, while the minute hand is controlled by a(n) _____ loop.
   a. outer, nested
   b. nested, outer

4. Which of the following can be used to code the mathematical expression 10^2?
   a. `pow(10.0, 2.0)`
   b. `pow(10.0, 2)`
   c. `pow(10, 2.0)`
   d. all of the above

The answers to the labs are located in Appendix A.



LAB 8-1 Stop and Analyze

The program shown in Figure 8-34 displays the total sales made in Region 1 and the total sales made in Region 2. Study the program and then answer the questions.

```
 1  //Lab8-1.cpp - displays each region's total sales
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  using namespace std;
 6
 7  int main()
 8  {
 9     //declare variables
10     int sales            = 0;
11     int region           = 1;   //counter
12     int totalRegionSales = 0;   //accumulator
13
```

Figure 8-34 Code for Lab 8-1 (*continues*)

(*continued*)

```
14     while (region < 3)
15     {
16        //get current region's first sales amount
17        cout << "First sales amount for Region "
18             << region << ": ";
19        cin >> sales;
20
21        while (sales > 0)
22        {
23           //add the sales amount to the total
24           //for the region
25           totalRegionSales += sales;
26           //get the next sales amount for the
27           //current region
28           cout << "Next sales amount for Region "
29                << region << ": ";
30           cin >> sales;
31        }  //end while
32
33        //display the current region's total sales
34        cout << "*****************Region " << region
35             << " sales: $" << totalRegionSales
36             << endl << endl;
37
38        //update the counter
39        region += 1;
40        ____________________          missing statement
41     }    //end while
42
43     system("pause");
44     return 0;
45  }  //end of main function
```

Figure 8-34 Code for Lab 8-1

QUESTIONS

1. How many pretest loops does the program contain? How many posttest loops does the program contain?
2. Which of the two loops is controlled by a counter? For what counter values will the loop's condition evaluate to true? What counter value will make the loop condition evaluate to false?
3. Which of the two loops is controlled by a sentinel value? What are the valid sentinel values for this loop?
4. The statement on Line 40 is missing from the program. To determine the missing statement, desk-check the program using sales amounts of 1000, 2000, and -1 for Region 1, and sales amounts of 400, 500, and -3 for Region 2. What is the missing statement and why is it necessary? How would you modify the comment on Line 38 so it documents what the additional statement does?
5. After entering the missing statement on Line 40, desk-check the program using the data from Question 4. What are the total sales for Region 1? What are the total sales for Region 2?

6. Follow the instructions for starting C++ and opening the Lab8-1.cpp file. The file is contained in either the Cpp6\Chap08\Lab8-1 Project folder or the Cpp6\Chap08 folder. Enter the missing statement on Line 40. Also modify the comment on Line 38. Run the program and then enter the data from Question 4. What does the program display as the total sales for each region? (The program's output should agree with the results of your desk-check from Question 5.)

7. Modify the program to use a `for` statement for the region loop and a `do while` statement for the sales loop. Save and then run the program. Enter the data from Question 4. What does the program display as the total sales for each region? (The program's output should agree with the results from Question 6.)



LAB 8-2 Plan and Create

In this lab, you will plan and create an algorithm for Mrs. Johansen. The problem specification is shown in Figure 8-35.

Problem specification

Last month, Mrs. Johansen began teaching multiplication to the students in her second grade class. She wants a program that displays one or more multiplication tables. A sample multiplication table is shown below. The *x* entries represent the number entered by the user and are called the multiplicand. The numbers 1 through 9 are called the multiplier. The *y* entries represent the product, which is the result of multiplying the multiplicand (*x*) by the multiplier (the numbers 1 through 9).

multiplicand

multiplier

product

Table format	Sample table using a multiplicand of 2
x * 1 = y	2 * 1 = 2
x * 2 = y	2 * 2 = 4
x * 3 = y	2 * 3 = 6
x * 4 = y	2 * 4 = 8
x * 5 = y	2 * 5 = 10
x * 6 = y	2 * 6 = 12
x * 7 = y	2 * 7 = 14
x * 8 = y	2 * 8 = 16
x * 9 = y	2 * 9 = 18

Figure 8-35 Problem specification for Lab 8-2

First, analyze the problem, looking for the output first and then for the input. In this case, Mrs. Johansen wants the program to display a multiplication table. The table will show the multiplicand entered by the user, the multipliers provided in the problem specification, and the products calculated by the program. Next, plan the algorithm. Recall that most algorithms begin with an instruction to enter the input items into the computer, followed by instructions that process the input items, typically including the items in one or more calculations. Most algorithms end with one or more instructions that display, print, or store the output items. Figure 8-36 shows the completed IPO chart for the multiplication table problem.

Input	Processing	Output
multiplicand	Processing items: multiplier (counter: 1 to 9) product Algorithm: 1. enter the first multiplicand 2. repeat while (the multiplicand is greater than or equal to 0) repeat for (multiplier from 1 to 9 in increments of 1) calculate the product by multiplying the multiplicand by the multiplier display the multiplicand, multiplier, and product, then move the cursor to the next line end repeat display a blank line between tables enter the next multiplicand end repeat	multiplication table (multiplicand * multiplier = product)

Figure 8-36 Completed IPO chart for the multiplication table problem

After completing the IPO chart, you then move on to the third step in the problem-solving process, which is to desk-check the algorithm. You will desk-check the algorithm in Figure 8-36 using multiplicands of 2 and 4, followed by a sentinel value of -1. Figure 8-37 shows the completed desk-check table. Notice that the products corresponding to a multiplicand of 2 agree with the amounts shown in the sample table in Figure 8-35.

multiplicand	multiplier	product
2	1	2
	2	4
	3	6
	4	8
	5	10
	6	12
	7	14
	8	16
	9	18
4	1	4
	2	8
	3	12
	4	16
	5	20
	6	24
	7	28
	8	32
	9	36
-1		

Figure 8-37 Completed desk-check table for the multiplication table algorithm

The fourth step in the problem-solving process is to code the algorithm into a program. You begin by declaring memory locations that will store the values of the input, processing, and output items. The multiplication table program will need three memory locations to store the multiplicand, multiplier, and product. You will store the multiplicand in a variable, because the user should be allowed to change its value during runtime. The multiplier also will be stored in a variable to allow its value to change from 1 to 9 in increments of 1. Finally, you will store the product in a variable, because its value will change based on the current values of the multiplicand and multiplier. The multiplicand, multiplier, and product will always be integers, so you will store them in `int` variables. Figure 8-38 shows the IPO chart information and corresponding C++ instructions.

IPO chart information	C++ instructions
Input	
multiplicand	`int multiplicand = 0;`
Processing	
multiplier (counter: 1 to 9)	this variable is created and initialized in the for clause
product	`int product = 0;`
Output	
multiplication table	contains the multiplicand, multiplier, and product
Algorithm	
1. enter the first multiplicand	`cout << "Multiplicand (negative number to end): ";` `cin >> multiplicand;`
2. repeat while (the multiplicand is greater than or equal to 0)	`while (multiplicand >= 0)` `{`
repeat for (multiplier from 1 to 9 in increments of 1)	`for (int multiplier = 1; multiplier < 10; multiplier += 1)` `{`
calculate the product by multiplying the multiplicand by the multiplier	`product = multiplicand * multiplier;`
display the multiplicand, multiplier, and product, then move the cursor to the next line	`cout << multiplicand << " * " << multiplier << " = " << product << endl;` `} //end for`
end repeat	
display a blank line between tables	`cout << endl;`
enter the next multiplicand	`cout << "Multiplicand (negative number to end): ";` `cin >> multiplicand;`
end repeat	`} //end while`

Figure 8-38 IPO chart information and C++ instructions for the multiplication table program

The fifth step in the problem-solving process is to desk-check the program. You begin by placing the names of the declared variables and named constants (if any) in a new desk-check table, along with their initial values. You then desk-check the remaining C++ instructions in order, recording in the desk-check table any changes made to the variables. Figure 8-39 shows the completed desk-check table for the multiplication table program. The results agree with those shown in the algorithm's desk-check table in Figure 8-37.

multiplicand	multiplier	product
0		0
2	1	2
	2	4
	3	6
	4	8
	5	10
	6	12
	7	14
	8	16
	9	18
4	1	4
	2	8
	3	12
	4	16
	5	20
	6	24
	7	28
	8	32
	9	36
-1		

Figure 8-39 Completed desk-check table for the multiplication table program

The final step in the problem-solving process is to evaluate and modify (if necessary) the program. Recall that you evaluate a program by entering its instructions into the computer and then using the computer to run (execute) it. While the program is running, you enter the same sample data used when desk-checking the program.

DIRECTIONS

Follow the instructions for starting your C++ development tool. Depending on the development tool you are using, you may need to create a new project; if so, name the project Lab8-2 Project and save it in the Cpp6\Chap08 folder. Enter the instructions shown in Figure 8-40 in a source file named Lab8-2.cpp. (Do not enter the line numbers.) Save the file in either the project folder or the Cpp6\Chap08 folder. Now follow the appropriate instructions for running the Lab8-2.cpp file. Test the program using multiplicands of 2 and 4, followed by a sentinel value. If necessary, correct any bugs (errors) in the program.



You also can write the `for` clause's *update* argument, which appears on Line 19 in Figure 8-40, as `multiplier = multiplier + 1`.

```
 1  //Lab8-2.cpp - displays a multiplication table
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  using namespace std;
 6
 7  int main()
 8  {
 9      //declare variables
10      int multiplicand = 0;
11      int product      = 0;
12
13      cout << "Multiplicand (negative number to end): ";
14      cin >> multiplicand;
15
16      while (multiplicand >= 0)
17      {
18            for (int multiplier = 1; multiplier < 10;
19                  multiplier += 1)
20            {
21                  product = multiplicand * multiplier;
22                  cout << multiplicand << " * "
23                       << multiplier << " = "
24                       << product << endl;
25            }     //end for
26
27            cout << endl;
28            cout << "Multiplicand (negative number to end): ";
29            cin >> multiplicand;
30      }     //end while
31
32      system("pause");
33      return 0;
34  }  //end of main function
```

if your C++ development tool does not require this statement, either omit it or make it a comment

Figure 8-40 Multiplication table program



LAB 8-3 Modify

If necessary, create a new project named Lab8-3 Project. Enter (or copy) the Lab8-2.cpp instructions into a new source file named Lab8-3.cpp. Change Lab8-2.cpp in the first comment to Lab8-3.cpp. Change both loops to posttest loops. Save and then run the program. Use the program to display the multiplication tables for the following multiplicands: 6, 9, and 2.



LAB 8-4 Desk-Check

Desk-check the code shown in Figure 8-41. What will the code display on the computer screen?

```
int number = 1;

while (number < 3)
{
      cout << number << ' ';
      for (int x = 1; x <= 4; x += 1)
            cout << number + x << ' ';
      //end for
      number += 1;
      cout << endl;
}     //end while
```

Figure 8-41 Code for Lab 8-4



LAB 8-5 Debug

Follow the instructions for starting C++ and opening the Lab8-5.cpp file. The file is contained in either the Cpp6\Chap08\Lab8-5 Project folder or the Cpp6\Chap08 folder. Debug the program.

Summary

- A repetition structure can be either a pretest loop or a posttest loop. In a pretest loop, the loop condition is evaluated *before* the instructions within the loop are processed. In a posttest loop, the evaluation occurs *after* the instructions within the loop are processed.
- The condition appears at the end of a posttest loop and determines whether the instructions within the loop body will be processed more than once. The loop's condition must result in either a true or false answer only. When the condition evaluates to true, the instructions listed in the loop body are processed again; otherwise, the loop is exited.
- You use the `do while` statement to code a posttest loop in C++. You can use either the `while` statement or the `for` statement to code a pretest loop in C++.
- Repetition structures can be nested, which means one loop (called the inner or nested loop) can be placed inside another loop (called the outer loop). For nested repetition structures to work correctly, the entire inner loop must be contained within the outer loop.

- You can use the built-in C++ `pow` function to raise a number to a power. The function returns the result as a `double` number.

Key Terms

Exponentiation—the process of raising a number to a power

Nested loop—a loop (repetition structure) contained entirely within another loop (repetition structure)

pow function—a built-in C++ function that raises a number to a power and then returns the result as a `double` number

Review Questions

1. The condition in the `do while` statement is evaluated _____ the instructions in the loop body are processed.

   a. after

   b. before

2. The instructions in the body of the _____ statement always are processed at least once during runtime.

   a. `do while`

   b. `for`

   c. `while`

   d. both a and b

3. It's possible that the instructions in the body of the _____ statement will not be processed during runtime.

   a. `do while`

   b. `for`

   c. `while`

   d. both b and c

4. What numbers will the following code display on the computer screen?

```
int x = 1;
do
{
    cout << x << endl;
    x = x + 1;
} while (x < 5);
```

   a. 0, 1, 2, 3, 4

   b. 0, 1, 2, 3, 4, 5

   c. 1, 2, 3, 4

   d. 1, 2, 3, 4, 5

5. What numbers will the following code display on the computer screen?

```
int x = 20;
do
{
    cout << x << endl;
    x = x - 4;
} while (x > 10);
```

a. 16, 12, 8

b. 16, 12

c. 20, 16, 12, 8

d. 20, 16, 12

6. What value of `x` causes the loop in Review Question 5 to end?

a. 0

b. 8

c. 10

d. 12

7. What numbers will the following code display on the computer screen?

```
int total = 1;
do
{
    cout << total << endl;
    total = total + 2;
} while (total <= 3);
```

a. 1, 2

b. 1, 3

c. 1, 3, 5

d. 0, 1, 3

8. What will the following code display on the computer screen?

```
for (int x = 1; x < 3; x = x + 1)
{
    for (int y = 1; y < 4; y = y + 1)
        cout << "*";
    //end for
    cout << endl;
} //end for
```

a.
```
***
***
```

b.
```
***
***
***
```

c.
```
**
**
**
```

d.
```
***
***
***
***
```

9. What number will the following code display on the computer screen?

```
int sum = 0;
int y = 0;
do
{
    for (int x = 1; x < 5; x = x + 1)
        sum = sum + x;
    //end for
    y = y + 1;
} while (y < 3);
cout << sum << endl;
```

a. 5

b. 8

c. 15

d. 30

10. Which of the following raises the number 6 to the third power?

a. `cube(6)`

b. `pow(3.0, 6)`

c. `pow(6.0, 3)`

d. none of the above

Exercises



Pencil and Paper

TRY THIS

1. Complete a desk-check table for the code shown in Figure 8-42. What will the code display on the computer screen? What value causes the nested loop to end? What value causes the outer loop to end? (The answers to TRY THIS Exercises are located at the end of the chapter.)

```
int nested = 1;
for (int outer = 1; outer <= 2; outer += 1)
{
      do    //begin loop
      {
            cout << nested;
            cout << "    ";
            nested += 1;
      }  while (nested < 4);
      cout << endl;
      nested = 1;
}  //end for
```

Figure 8-42

2. Write a C++ `while` clause that stops a posttest loop when the value in the `quantity` variable is less than the number 0. (The answers to TRY THIS Exercises are located at the end of the chapter.) — TRY THIS

3. Rewrite the code shown in Figure 8-42 so it uses the `while` statement for the outer loop and the `for` statement for the nested loop. — MODIFY THIS

4. Rewrite the code shown in Figure 8-42 to use the `do while` statement for the outer loop. — MODIFY THIS

5. Write a C++ `while` clause that processes a posttest loop's instructions as long as the value in the `inStock` variable is greater than the value in the `reorder` variable. — INTRODUCTORY

6. Write a C++ `while` clause that stops a posttest loop when the value in a `char` variable named `letter` is anything other than the letter Y (in either uppercase or lowercase). — INTRODUCTORY

7. Write an assignment statement that raises the number 2 to the 25th power and then assigns the result to a `double` variable named `answer`. — INTRODUCTORY

8. A program declares an `int` variable named `oddNum` and initializes it to 1. Write the C++ code to display the odd integers 1, 3, 5, 7, and 9 on separate lines on the computer screen. Use the `do while` statement. — INTERMEDIATE

9. Write the code to display a table consisting of four rows and six columns. The first column should contain the numbers 1 through 4 raised to the first power. The second column should contain the result of raising the number in the first column to the second power. The third column should contain the result of raising the number in the first column to the third power, and so on. The table will look similar to the one shown in Figure 8-43. Use two `for` statements: one to keep track of the numbers 1 through 4, and the other to keep track of the powers (1 through 6). — INTERMEDIATE

1	1	1	1	1	1
2	4	8	16	32	64
3	9	27	81	243	729
4	16	64	256	1024	4096

Figure 8-43

INTERMEDIATE

10. Rewrite the code from Pencil and Paper Exercise 9 to use two `while` statements.

INTERMEDIATE

11. Rewrite the code from Pencil and Paper Exercise 9 to use two `do while` statements.

INTERMEDIATE

12. Rewrite the code from Pencil and Paper Exercise 9 to use a `while` statement in the outer loop.

INTERMEDIATE

13. Rewrite the code from Pencil and Paper Exercise 9 to use the `do while` statement in the nested loop.

SWAT THE BUGS

14. The code shown in Figure 8-44 should display three rows of asterisks on the computer screen. The first row should contain one asterisk. The second row should contain two asterisks, and the third row should contain three asterisks. However, the code is not working correctly; it displays three asterisks on each of the three rows. Debug the code.

```
for (int row = 1; row < 4; row += 1)
{
     for (int asterisks = 1; asterisks <= 3; asterisks += 1)
          cout << '*';
     //end for
     cout << endl;
}    //end for
```

Figure 8-44



Computer

TRY THIS

15. In this exercise, you create a program that uses two `for` statements to display the pattern of asterisks shown in Figure 8-45. If necessary, create a new project named TryThis15 Project. Enter the C++ instructions into a source file named TryThis15.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Save and then run the program. (The answers to TRY THIS Exercises are located at the end of the chapter.)

```
**
****
******
********
**********
```

Figure 8-45

16. In this exercise, you create a program that uses two `while` statements to display the pattern of asterisks shown in Figure 8-46. If necessary, create a new project named TryThis16 Project. Enter the C++ instructions into a source file named TryThis16.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Save and then run the program. (The answers to TRY THIS Exercises are located at the end of the chapter.)

TRY THIS

```
*********
********
*******
******
*****
****
***
**
*
```

Figure 8-46

17. In this exercise, you modify the code from Computer Exercise 16. If necessary, create a new project named ModifyThis17 Project. Enter (or copy) the TryThis16.cpp instructions into a new source file named ModifyThis17.cpp. Change TryThis16.cpp in the first comment to ModifyThis17.cpp. Replace the two `while` statements with `do while` statements. Save and then run the program.

MODIFY THIS

18. The savings calculator program is shown in Figure 8-29 in the chapter. If necessary, create a new project named Introductory18 Project. Enter the C++ instructions from the figure into a source file named Introductory18.cpp. Change the filename in the first comment. Save and then run the program. The output is shown in Figure 8-30 in the chapter. Replace the `do while` statement with a `for` statement, and then save and run the program again.

INTRODUCTORY

19. The modified savings calculator program is shown in Figure 8-32 in the chapter. If necessary, create a new project named Introductory19 Project. Enter the C++ instructions from the figure into a source file named Introductory19.cpp. Change the filename in the first comment. Save and then run the program. The output is shown in Figure 8-33 in the chapter. Replace the `for` statement with a `while` statement, and then save and run the program again.

INTRODUCTORY

20. In this exercise, you modify the code from Lab 8-2. If necessary, create a new project named Intermediate20 Project. Enter (or copy) the C++ instructions from the Lab8-2.cpp file into a source file named Intermediate20.cpp. Change the filename in the first comment. Replace both pretest loops with posttest loops. Save and then run the program.

INTERMEDIATE

21. The modified savings calculator program is shown in Figure 8-32 in the chapter. If necessary, create a new project named Intermediate21

INTERMEDIATE

Project. Enter the C++ instructions from the figure into a source file named Intermediate21.cpp. Modify the program to allow the user to enter the principal. Use a sentinel value to end the program. Save and then run the program. Test the program using the following principals, followed by your sentinel value: 1000, 3000, and 10000.

INTERMEDIATE

22. Write the code to display a table consisting of four rows and 11 columns. The first column should contain the numbers 1 through 4. The second and subsequent columns should contain the result of multiplying the number in the first column by the numbers 0 through 9. The table will look similar to the one shown in Figure 8-47. (Don't be concerned about the alignment of the numbers in each column.) Use two `for` statements. If necessary, create a new project named Intermediate22 Project. Enter your C++ instructions into a source file named Intermediate22.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Save and then run the program.

1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36

Figure 8-47

INTERMEDIATE

23. The payroll manager at Kenton Incorporated wants a program that allows him to enter an unknown number of payroll amounts for each of three stores: Store 1, Store 2, and Store 3. The program should calculate the total payroll and then display the result on the screen.

    a. Create an IPO chart for the problem, and then desk-check the algorithm using 23000 and 15000 as the payroll amounts for Store 1; 12000, 16000, 34000, and 10000 for Store 2; and 64000, 12000, and 70000 for Store 3.

    b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 8-38. Then code the algorithm into a program.

    c. Desk-check the program using the same data used to desk-check the algorithm.

    d. If necessary, create a new project named Intermediate23 Project. Enter your C++ instructions into a source file named Intermediate23.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

    e. Save and then run the program. Test the program using the same data used to desk-check the program.

INTERMEDIATE

24. In this exercise, you modify the program from Lab 7-2 in Chapter 7. If necessary, create a new project named Intermediate24 Project. Copy the instructions from the Lab7-2.cpp file into a source file named Intermediate24.cpp. (Alternatively, you can enter the instructions

from Figure 7-48 into the Intermediate24.cpp file.) Change the filename in the first comment. Modify the program to allow Professor Chang to display more than one student's total points and grade. Save and then run the program. Test the program appropriately.

25. At the beginning of every year, Khalid receives a raise on his previous year's salary. He wants a program that calculates and displays the amount of his annual raises for the next three years, using rates of 3%, 4%, 5%, and 6%. The program should end when Khalid enters a sentinel value as the salary. ADVANCED

    a. Create an IPO chart for the problem, and then desk-check the algorithm using annual salaries of $30000 and $50000.

    b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 8-38. Then code the algorithm into a program.

    c. Desk-check the program using the same data used to desk-check the algorithm.

    d. If necessary, create a new project named Advanced25 Project. Enter your C++ instructions into a source file named Advanced25.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

    e. Save and then run the program. Test the program using the same data used to desk-check the program.

26. In this exercise, you modify the program from Computer Exercise 15. The modified program will allow the user to specify the outer loop's ending value and its increment value. The ending value determines the maximum number of asterisks to display. The increment value determines the number of asterisks to repeat. ADVANCED

    a. If necessary, create a new project named Advanced26 Project. Enter (or copy) the TryThis15.cpp instructions into a new source file named Advanced26.cpp. Change TryThis15.cpp in the first comment to Advanced26.cpp. Make the appropriate modifications to the program.

    b. Save and then run the program. Test the program by entering the numbers 4 and 1 as the maximum number of asterisks and increment value, respectively. The program should display four rows of asterisks as follows: one asterisk, two asterisks, three asterisks, and four asterisks.

    c. Run the program again. This time, enter the numbers 9 and 3 as the maximum number of asterisks and increment value, respectively. The program should display three rows of asterisks as follows: three asterisks, six asterisks, and nine asterisks.

27. In this exercise, you modify the program from Computer Exercise 16. The modified program will allow the user to display the asterisks using one of two different patterns. Pattern 1 contains nine rows of asterisks as follows: nine asterisks, eight asterisks, seven asterisks, six asterisks, five asterisks, four asterisks, three asterisks, two asterisks, ADVANCED

and one asterisk. Pattern 2 contains nine rows of asterisks as follows: one asterisk, two asterisks, three asterisks, four asterisks, five asterisks, six asterisks, seven asterisks, eight asterisks, and nine asterisks.

a. If necessary, create a new project named Advanced27 Project. Enter (or copy) the TryThis16.cpp instructions into a new source file named Advanced27.cpp. Change TryThis16.cpp in the first comment to Advanced27.cpp. Make the appropriate modifications to the program.

b. Save and then run the program. Display the asterisks using the first pattern, and then display them using the second pattern.

ADVANCED

28. The modified savings calculator program is shown in Figure 8-32 in the chapter. If necessary, create a new project named Advanced28 Project. Enter the C++ instructions from the figure into a source file named Advanced28.cpp. Modify the program to allow the user to enter the principal, the minimum and maximum interest rates, and the number of years. Save and then run the program. Test the program appropriately. Include the following values in your test data: 1000, .04, .06, and 5.

SWAT THE BUGS

29. Follow the instructions for starting C++ and opening the SwatTheBugs29.cpp file. The file is contained in either the Cpp6\Chap08\SwatTheBugs29 Project folder or the Cpp6\Chap08 folder. Debug the program.

Answers to TRY THIS Exercises



Pencil and Paper

1. See Figure 8-48. The number 4 causes the nested loop to end. The number 3 causes the outer loop to end.

```
nested          outer
1               1
2               2
3               3
4
1
2
3
4
1
```

The program will display the following output:
1 2 3
1 2 3

Figure 8-48

2. `} while (quantity >= 0);`

Computer

15. See Figure 8-49.

```
 1  //TryThis15.cpp - displays a pattern of asterisks
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  using namespace std;
 6
 7  int main()
 8  {
 9     for (int outer = 2; outer < 11; outer += 2)
10     {
11          for (int nested = 1; nested <= outer; nested += 1)
12               cout << '*';
13          //end for
14          cout << endl;
15     }    //end for
16
17     system("pause");
18     return 0;
19  }  //end of main function
```

Figure 8-49

16. See Figure 8-50.

```
 1  //TryThis16.cpp - displays a pattern of asterisks
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  using namespace std;
 6
 7  int main()
 8  {
 9     int outer  = 9;
10     int nested = 1;
11
12     while (outer > 0)
13     {
14          while (nested <= outer)
15          {
16               cout << '*';
17               nested += 1; //update counter
18          }    //end while
19          outer -= 1;     //update counter
20          nested = 1;     //reset counter
21          cout << endl;
22     }    //end while
23
24     system("pause");
25     return 0;
26  }  //end of main function
```

Figure 8-50

CHAPTER 9

Value-Returning Functions

After studying Chapter 9, you should be able to:

- Use the `sqrt` function to return the square root of a number
- Generate random numbers
- Create and invoke a function that returns a value
- Pass information *by value* to a function
- Write a function prototype
- Understand a variable's scope and lifetime

Functions

As you learned in Chapter 4, a function is a block of code that performs a task. Every C++ program contains at least one function, which is named `main`; however, most C++ programs contain many functions. Some of the functions used in a program are built into the C++ language. The code for these **built-in functions** resides in C++ libraries, which are special files that come with the C++ language. Examples of built-in functions with which you already are familiar include `pow` and `toupper`. Other program functions, like `main`, are created by the programmer. These functions often are referred to as **program-defined functions**, because the function definitions are contained in the program itself rather than in a different file. But why would a programmer need more than the `main` function? One reason is to avoid the duplication of code. For instance, assume that the same task needs to be performed in more than one section of a program. Rather than enter the appropriate code in each of those sections, it is more efficient for the programmer to enter the code once, in a function. Any section in the program can then call (or invoke) the function to perform the required task. Program-defined functions also allow large and complex programs, which typically are written by a team of programmers, to be broken into small and manageable tasks. Each member of the team is assigned one or more tasks to code as a function. When each programmer completes his or her function, all of the functions are gathered together into one program. In other words, program-defined functions allow more than one programmer to work on a program at the same time, decreasing the time it takes to write the program. Typically, a program's `main` function is responsible for calling (or invoking) each of the other program-defined functions. However, any program-defined function can call any other program-defined or built-in function. All program-defined and built-in functions are categorized as either value-returning functions or void functions. Value-returning functions return a value, whereas void functions do not return a value. You will learn about value-returning functions in this chapter. Void functions are covered in Chapter 10.



In some programming languages, functions are called methods, subroutines, or procedures.

All **value-returning functions**, whether built-in or program-defined, perform a task and then return precisely one value after the task is completed. The built-in value-returning `toupper` function, for example, temporarily converts a character to uppercase and then returns the result. The built-in value-returning `pow` function, on the other hand, returns the result of raising a number to a specified power. In almost all cases, a value-returning function returns its one value to the statement from which it was called (invoked). One exception is the `main` function, which returns its one value to the operating system. Typically, the statement that invokes a function assigns the return value to a variable. However, it also may use the return value in a calculation or comparison; or it simply may display the return value. In the first part of this chapter, you will learn how to use several built-in value-returning functions. Later in the chapter, you will learn how to create program-defined value-returning functions. At this point, it's important to point out that functions are one of the more challenging topics for beginning programmers. Therefore, don't be concerned if you don't understand everything right away. If you still feel overwhelmed by the end of the chapter, try reading the chapter again, paying particular attention to the examples and programs shown in the figures.

The Hypotenuse Program

Figure 9-1 shows the problem specification and IPO chart for the hypotenuse program. (The flowchart for this program is contained in the Ch9Flowcharts.pdf file, which is located in the Cpp6\Chap09 folder.) The program calculates and displays the length of a right triangle's hypotenuse, which is the longest side of the triangle. The figure also includes an example of using the Pythagorean Theorem to calculate the length. The theorem requires raising a number to the second power and also taking the square root of a number. You already know how to square a number using the C++ built-in `pow` function. In the next section, you will learn how to use the built-in `sqrt` function to find the square root of a number. Both functions are value-returning functions because they return a value after performing their assigned task.



Raising a number to the second power is the same as squaring a number.

Problem specification
Create a program that calculates and displays the length of the hypotenuse of a right triangle, given the lengths of the triangle's two adjacent sides (*side a* and *side b*). You can calculate the length using the Pythagorean Theorem, which indicates that the length of the hypotenuse is equal to the square root of the sum of the squares of the lengths of a right triangle's two adjacent sides. In other words, the hypotenuse's length is equal to the square root of the following sum: (*side a length*)^2 + (*side b length*)^2.

Example	*side a length* is 10 and *side b length* is 24
1. square *side a length*	10 * 10 = 100
2. square *side b length*	24 * 24 = 576
3. sum the squares from Steps 1 and 2	100 + 576 = 676
4. find the square root of the sum from Step 3	26 — length of the hypotenuse

Input
side a length
side b length

Processing
Processing items:
sum of the squares

Algorithm:
1. enter side a length and side b length
2. calculate the sum of the squares
   = (side a length)^2 + (side b length)^2
3. calculate the hypotenuse length by finding the square root of the sum of the squares
4. display the hypotenuse length

Output
hypotenuse length

Figure 9-1 Problem specification, calculation example, and IPO chart for the hypotenuse program

Finding the Square Root of a Number

In Chapter 8, you learned that you can find the square root of a number by raising the number to the .5 power, like this: `pow(100, .5)`. Although you can use the `pow` function to determine a square root, C++ provides the `sqrt` function specifically for that purpose. The **sqrt function** is a built-in value-returning function that calculates a number's square root and then returns the result as a `double` number. The `sqrt` function's code is

contained in the cmath library file. Therefore, a program must contain the `#include <cmath>` directive in order to use the `sqrt` function. The directive tells the C++ compiler the location of the function's code. The `sqrt` function's syntax is shown in Figure 9-2. Recall from Chapter 5 that an item within parentheses in a function's syntax—in this case, *x*—is called an argument; more specifically, it is called an actual argument. An **actual argument** represents information that the function needs to perform its task. The information is passed to the function when the function is invoked. Invoking a function also is referred to as calling a function. You call a function simply by including its name and actual arguments (if any) in a program statement. The `sqrt` function contains one actual argument (*x*), because it requires only one item of information: the number whose square root you want to find. The *x* actual argument must have either the `double` or `float` data type. Also included in Figure 9-2 are examples of using the `sqrt` function. (For clarity, the variable declaration statements are included in the examples.) The assignment statement in Example 1 calls (invokes) the `sqrt` function, passing it the `double` number 100.0. The function calculates the number's square root and then returns the answer to the assignment statement, which assigns the answer to the `squareRoot` variable. Similarly, the second `cout` statement in Example 2 calls the `sqrt` function, passing it the `double` number stored in the `num` variable. The function calculates the number's square root and then returns the answer to the `cout` statement, which displays the answer on the computer screen.



Recall from Chapter 4 that a `#include` directive provides a convenient way to merge the source code from one file with the source code in another file, without having to retype the code.

HOW TO Use the `sqrt` Function

Syntax

sqrt(*x***)** — requires the `#include <cmath>` directive

Example 1

```
double squareRoot = 0.0;
squareRoot = sqrt(100.0);
```

The `sqrt` function finds the square root of the `double` number 100.0 and then returns the result (the `double` number 10.0) to the assignment statement, which assigns the return value to the `squareRoot` variable.

Example 2

```
double num = 0.0;
cout << "Enter a number: ";
cin >> num;
cout << sqrt(num);
```

The `sqrt` function finds the square root of the `double` number stored in the `num` variable and then returns the result to the `cout` statement, which displays the return value on the computer screen.

Figure 9-2 How to use the `sqrt` function

Figure 9-3 shows the IPO chart information and corresponding C++ instructions for the hypotenuse program, and Figure 9-4 shows all of the program's code. The `#include <cmath>` directive on Line 6 in the code is required because the program uses the `sqrt` function. The directive tells the C++ compiler to include the contents of the cmath file, which contains the `sqrt` function's code, in the current program. The `sqrt` function appears in the assignment statement on Line 26. The assignment statement calls the function, passing it the value stored in the `sumSqrs` variable. When the computer encounters the assignment statement in the code, it temporarily leaves the `main` function to process the `sqrt` function's code. The `sqrt` function calculates the square root of the value passed to it and then returns the value to the statement that called it. In this case, it returns the value to the assignment statement on Line 26. The assignment statement assigns the return value to the `double hypotenuse` variable. A sample run of the hypotenuse program is shown in Figure 9-5.

IPO chart information	C++ instructions
Input	
side a length	`double sideA = 0.0;`
side b length	`double sideB = 0.0;`
Processing	
sum of the squares	`double sumSqrs = 0.0;`
Output	
hypotenuse length	`double hypotenuse = 0.0;`
Algorithm	

1. enter side a length and side b length	`cout << "Side a length: ";`
	`cin >> sideA;`
	`cout << "Side b length: ";`
	`cin >> sideB;`

2. calculate the sum of the squares = (side a length)^2 + (side b length)^2	`sumSqrs = pow(sideA, 2) +` `pow(sideB, 2);`
3. calculate the hypotenuse length by finding the square root of the sum of the squares	`hypotenuse = sqrt(sumSqrs);`
4. display the hypotenuse length	`cout << "Hypotenuse length: "` `<< hypotenuse << endl;`

Figure 9-3 IPO chart information and C++ instructions for the hypotenuse program

```
 1  //Hypotenuse.cpp - displays the length of the
 2  //hypotenuse of a right triangle
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  #include <cmath>
 7  using namespace std;
 8
 9  int main()
10  {
11
12     //declare variables
13     double sideA      = 0.0;
14     double sideB      = 0.0;
15     double sumSqrs    = 0.0;
16     double hypotenuse = 0.0;
17
18     //get lengths of two sides
19     cout << "Side a length: ";
20     cin >> sideA;
21     cout << "Side b length: ";
22     cin >> sideB;
23
24     //calculate the length of the hypotenuse
25     sumSqrs = pow(sideA, 2) + pow(sideB, 2);
26     hypotenuse = sqrt(sumSqrs);
27
28     //display the length of the hypotenuse
29     cout << "Hypotenuse length: "
30          << hypotenuse << endl;
31
32     system("pause");
33     return 0;
34  }  //end of main function
```

required for the sqrt function

uses the sqrt function

your C++ development tool may not require this statement

Figure 9-4 Hypotenuse program

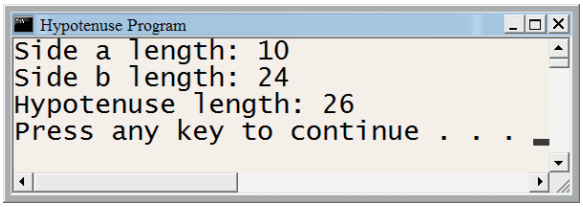


Figure 9-5 Sample run of the hypotenuse program

The Random Addition Problems Program

Figure 9-6 shows the problem specification and IPO chart for the random addition problems program. (The flowchart for this program is contained in the Ch9Flowcharts.pdf file, which is located in the Cpp6\Chap09 folder.) The algorithm requires the computer to generate two random integers from 1 to 10, inclusive. You will learn how to generate random integers in the next section.

Problem specification
Create a program that displays five random addition problems, one at a time, on the computer screen. Each problem should be displayed as a question, like this: *What is the sum of x + y?*. The *x* and *y* in the question represent numbers from 1 to 10, inclusive. After displaying the question, the program should allow the user to enter the answer. It then should compare the user's answer with the correct answer. If the user's answer matches the correct answer, the program should display the "Correct!" message. Otherwise, it should display the "Sorry, the answer is" message followed by the correct answer and a period.

Input	Processing	Output
user's answer	Processing items: first random number (1 to 10) second random number (1 to 10) counter (1 to 5) correct answer Algorithm: 1. initialize the random number generator 2. repeat for (counter from 1 to 5) generate the first random number generate the second random number calculate the correct answer by adding together the first random number and second random number display the addition problem enter the user's answer if (user's answer matches correct answer) display "Correct!" message else display "Sorry, the answer is" message followed by the correct answer and a period end if display two blank lines end repeat	addition problem message

Figure 9-6 Problem specification and IPO chart for the random addition problems program

Generating Random Integers

Many programs require the use of random numbers. Examples of such programs include game programs, lottery programs, and programs used to practice elementary math skills. Most programming languages provide a **pseudo-random number generator**, which is a device that produces a sequence of numbers that meet certain statistical requirements for randomness. Pseudo-random numbers are chosen with equal probability from a finite set of numbers. The chosen numbers are not completely random, because a definite mathematical algorithm is used to select them. However, they are sufficiently random for practical purposes. The random number generator in C++ is a built-in value-returning function named `rand`. The **rand function** returns an integer that is greater than or equal to zero but

less than or equal to the value stored in the RAND_MAX constant, which is one of many constants built into the C++ language. Although the value of **RAND_MAX** varies with different systems, its value is always at least 32767. Figure 9-7 shows the `rand` function's syntax and includes examples of using the function to generate random integers. The `rand` function's syntax does not contain any actual arguments, because the function does not require any information to perform its task. However, notice that the parentheses after the function's name are required.



The `cout << RAND_MAX;` statement will display the value of RAND_MAX on your computer system.



In C++, every function's name is followed by a set of parentheses, which may or may not contain actual arguments.

HOW TO Use the `rand` Function

Syntax

rand()

Example 1

```
int randomNum = 0;
randomNum = rand();
```

The `rand` function generates a random integer that is greater than or equal to zero but less than or equal to RAND_MAX. It then returns the random integer to the assignment statement, which assigns the random integer to the `randomNum` variable.

Example 2

```
cout << rand();
```

The `rand` function generates a random integer that is greater than or equal to zero but less than or equal to RAND_MAX. It then returns the random integer to the `cout` statement, which displays the random integer on the computer screen.

Example 3

```
int tripleNum = 0;
tripleNum = rand() * 3;
```

The `rand` function generates a random integer that is greater than or equal to zero but less than or equal to RAND_MAX. It then returns the random integer to the assignment statement, which multiplies the random integer by three and assigns the result to the `tripleNum` variable.

Figure 9-7 How to use the `rand` function

Most programs that use random numbers require the numbers to be within a range that is more specific than zero through RAND_MAX. For example, a program that simulates rolling dice will require integers from one through six only. A program that displays arithmetic problems for elementary school students, on the other hand, may require integers from 10 through 100. Figure 9-8 shows the syntax of an expression that you can use to specify the desired range of integers. In the syntax, *lowerBound* and *upperBound* are the lowest integer and highest integer, respectively, in the range. Also included in Figure 9-8 are examples of using the expression in a C++ statement. The expression in Example 1 will produce integers in the range of one through six, whereas the expression in Example 2 will produce integers from 10 through 100. Figure 9-8 also shows how the computer evaluates the expressions using sample values generated by

the rand function. The first sample value for Example 1 is 27. When processing the 1 + 27 % (6 - 1 + 1) expression, the computer first evaluates the part of the expression within parentheses; in this case, 6 - 1 + 1 evaluates to 6. The expression now becomes 1 + 27 % 6. The modulus operation in the expression is evaluated next. The modulus operator (%) tells the computer to divide the number 27 by the number 6 and then find the remainder; in this case, the remainder is 3. The expression now becomes 1 + 3. Finally, the computer adds the number 1 to the number 3, giving 4. In other words, the expression in Example 1 evaluates to the integer 4 when the rand value is 27. As Figure 9-8 indicates, the expression will evaluate to the integers 3 and 1 when the rand value is 8 and 324, respectively. Notice that the three random integers (4, 3, and 1) are within the range of one through six. Now look closely at the sample rand values used in Example 2; the first sample value is 352. When processing the 10 + 352 % (100 - 10 + 1) expression, the computer first evaluates the part of the expression within parentheses; in this case, 100 - 10 + 1 evaluates to 91. The expression now becomes 10 + 352 % 91. The modulus operator (%) in the expression tells the computer to divide the number 352 by the number 91 and then find the remainder; in this case, the remainder is 79. The expression now becomes 10 + 79. Finally, the computer adds the number 10 to the number 79, giving 89. Therefore, the expression in Example 2 evaluates to the integer 89 when the rand value is 352. The expression will evaluate to the integers 14 and 53 when the rand value is 4 and 2500, respectively. Notice that, in this case, the three random integers (89, 14, and 53) are within the range of 10 through 100.

Recall from Chapter 4 that the modulus operator divides two integers and then returns the remainder as an integer.

You also can write the expressions in Figure 9-8 as 1 + rand() % 6 and 10 + rand() % 91. However, including the *lowerBound* and *upperBound* values within the parentheses makes the expression clearer and more self-documenting.

HOW TO Generate Random Integers within a Specific Range

Syntax

lowerBound **+ rand() % (***upperBound* **–** *lowerBound* **+ 1)**

Example 1

```
cout << 1 + rand() % (6 - 1 + 1);
```

displays a random integer from 1 through 6 on the computer screen

rand value: 27	1 + 27 % (6 - 1 + 1)
6 - 1 + 1 is evaluated first and results in 6	1 + 27 % 6
27 % 6 is evaluated next and results in 3	1 + 3
1 + 3 is evaluated last and results in 4	4
rand value: 8	1 + 8 % (6 - 1 + 1)
6 - 1 + 1 is evaluated first and results in 6	1 + 8 % 6
8 % 6 is evaluated next and results in 2	1 + 2
1 + 2 is evaluated last and results in 3	3
rand value: 324	1 + 324 % (6 - 1 + 1)
6 - 1 + 1 is evaluated first and results in 6	1 + 324 % 6
324 % 6 is evaluated next and results in 0	1 + 0
1 + 0 is evaluated last and results in 1	1

Figure 9-8 How to generate random integers within a specific range *(continues)*

(continued)

Example 2

```
int num = 0;
num = 10 + rand() % (100 - 10 + 1);
```

assigns a random integer from 10 through 100 to the `num` variable

`rand` value: 352	`10 + 352 % (100 - 10 + 1)`
`100 - 10 + 1` is evaluated first and results in 91	`10 + 352 % 91`
`352 % 91` is evaluated next and results in 79	`10 + 79`
`10 + 79` is evaluated last and results in 89	`89`
`rand` value: 4	`10 + 4 % (100 - 10 + 1)`
`100 - 10 + 1` is evaluated first and results in 91	`10 + 4 % 91`
`4 % 91` is evaluated next and results in 4	`10 + 4`
`10 + 4` is evaluated last and results in 14	`14`
`rand` value: 2500	`10 + 2500 % (100 - 10 + 1)`
`100 - 10 + 1` is evaluated first and results in 91	`10 + 2500 % 91`
`2500 % 91` is evaluated next and results in 43	`10 + 43`
`10 + 43` is evaluated last and results in 53	`53`

Figure 9-8 How to generate random integers within a specific range

You should initialize the random number generator in each program in which it is used. Otherwise, it will generate the same series of numbers each time the program is executed. Typically, the initialization task is performed at the beginning of the program. You initialize the random number generator using the **srand function**. Like the `rand` function, the `srand` function is a built-in C++ function. However, unlike the `rand` function, the `srand` function is a void function, which means it does not return a value. You will learn more about void functions in Chapter 10. Figure 9-9 shows the syntax of the `srand` function and includes examples of using the function. The *seed* actual argument in the syntax is an integer that represents the starting point for the random number generator. The computer uses the starting point (or seed) in the mathematical algorithm it employs when selecting the random numbers. You can have the user enter the seed, as shown in Example 1 in Figure 9-9. However, a more common way to initialize the random number generator is to use the C++ `time` function as the seed; this is shown in Examples 2 and 3. The **time function** is a built-in value-returning function that returns the current time (according to your computer system's clock) as seconds elapsed since midnight on January 1, 1970. However, because the `time` function returns a `time_t` object, you will need to use a type cast to convert the function's return value to an integer, as shown in Examples 2 and 3. In both examples, the `time` function is passed one actual argument: the number 0. Using the `time` function as the `srand` function's seed ensures that the random number generator is initialized with a unique integer each time the program is executed. The unique integer will produce a unique series of random numbers. To use the `time` function in a program, the program must contain the `#include <ctime>` directive.

HOW TO Use the `srand` Function

Syntax

srand(*seed***)**

Example 1

```
int x = 0;
cout << "Enter an integer: ";
cin >> x;
srand(x);
cout << rand() << endl;
cout << rand() << endl;
```

The `srand` function initializes the random number generator using the integer entered by the user. The `cout` statements display two random integers on the computer screen. The random integers will be greater than or equal to zero but less than or equal to `RAND_MAX`.

Example 2

the `time` function requires the `#include <ctime>` directive

```
srand(static_cast<int>(time(0)));
cout << rand() << endl;
cout << rand() << endl;
```

The `srand` function initializes the random number generator using the value returned by the `time` function after it has been converted to the `int` data type. The `cout` statements display two random integers on the computer screen. The random integers will be greater than or equal to zero but less than or equal to `RAND_MAX`.

Example 3

the `time` function requires the `#include <ctime>` directive

```
int randNum = 0;
srand(static_cast<int>(time(0)));
randNum = 1 + rand() % (10 - 1 + 1);
```

The `srand` function initializes the random number generator using the value returned by the `time` function after it has been converted to the `int` data type. The assignment statement assigns a random integer to the `randNum` variable. The random integer will be greater than or equal to one but less than or equal to 10.

Figure 9-9 How to use the `srand` function

Figure 9-10 shows the IPO chart information and corresponding C++ instructions for the random addition problems program, and Figure 9-11 shows all of the program's code. The `#include <ctime>` directive appears on Line 9 in the code and is required when using the `time` function. The statement on Line 21 uses the `time` and `srand` functions to initialize the random number generator. The `rand` function appears in the assignment statements on Lines 28 and 29; each assigns a random integer from one through 10 to a variable. A sample run of the random addition problems program is shown in Figure 9-12.

IPO chart information	C++ instructions
Input	
user's answer	`int userAnswer = 0;`
Processing	
first random number (1 to 10)	`int num1 = 0;`
second random number (1 to 10)	`int num2 = 0;`
counter (1 to 5)	this variable is created and initialized in the for clause
correct answer	`int correctAnswer = 0;`
Output	
addition problem	this contains string literal constants and the num1 and num2 variables
message	this is one of two messages composed of either a string literal constant or string literal constants and the correctAnswer variable
Algorithm	
1. initialize the random number generator	`srand(static_cast<int>(time(0)));`
2. repeat for (counter from 1 to 5)	`for (int x = 1; x < 6; x += 1)`
	`{`
generate the first random number	`num1 = 1 + rand() % (10 - 1 + 1);`
generate the second random number	`num2 = 1 + rand() % (10 - 1 + 1);`
calculate the correct answer by adding together the first random number and second random number	`correctAnswer = num1 + num2;`
display the addition problem	`cout << "What is the sum of "` `<< num1 << " + " << num2 << "? ";`
enter the user's answer	`cin >> userAnswer;`
if (user's answer matches correct answer)	`if (userAnswer == correctAnswer)`
display "Correct!" message	`cout << "Correct!";`
else	`else`
display "Sorry, the answer is" message followed by the correct answer and a period	`cout << "Sorry, the correct answer is " << correctAnswer << ".";`
end if	`//end if`
display two blank lines	`cout << endl << endl;`
end repeat	`}  //end for`

Figure 9-10 IPO chart information and C++ instructions for the random addition problems program

```
 1  //Random Addition.cpp
 2  //Displays random addition problems
 3  //Allows the user to enter the answer and then
 4  //displays a message that indicates whether the
 5  //user's answer is correct or incorrect
 6  //Created/revised by <your name> on <current date>
 7
 8  #include <iostream>
 9  #include <ctime>
10  using namespace std;
11
12  int main()
13  {
14      //declare variables
15      int num1          = 0;
16      int num2          = 0;
17      int correctAnswer = 0;
18      int userAnswer    = 0;
19
20      //initialize rand function
21      srand(static_cast<int>(time(0)));
22
23      for (int x = 1; x < 6; x += 1)
24      {
25            //generate two random integers
26            //from 1 through 10, then
27            //calculate the sum
28            num1 = 1 + rand() % (10 - 1 + 1);
29            num2 = 1 + rand() % (10 - 1 + 1);
30            correctAnswer = num1 + num2;
31
32            //display addition problem and get user's answer
33            cout << "What is the sum of " << num1
34                 << " + " << num2 << "? ";
35            cin >> userAnswer;
36
37            //detemine whether user's answer is correct
38            if (userAnswer == correctAnswer)
39                 cout << "Correct!";
40            else
41                 cout << "Sorry, the correct answer is "
42                      << correctAnswer << ".";
43            //end if
44            cout << endl << endl;
45      }     //end for
46
47      system("pause");
48      return 0;
49  }  //end of main function
```

required for the `time` function

uses the `srand` and `time` functions

uses the `rand` function

your C++ development tool may not require this statement

Figure 9-11 Random addition problems program

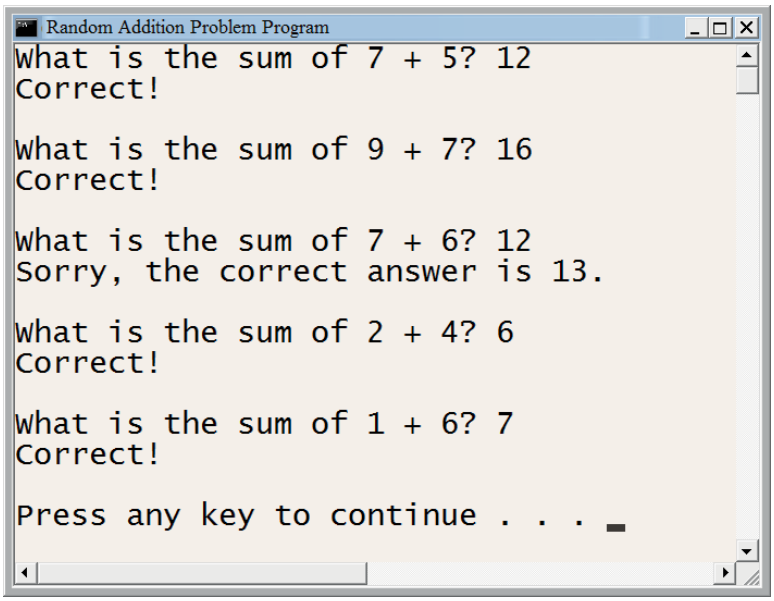

Random Addition Problem Program
What is the sum of 7 + 5? 12
Correct!
What is the sum of 9 + 7? 16
Correct!
What is the sum of 7 + 6? 12
Sorry, the correct answer is 13.
What is the sum of 2 + 4? 6
Correct!
What is the sum of 1 + 6? 7
Correct!
Press any key to continue . . .

Figure 9-12 Sample run of the random addition problems program

Mini-Quiz 9-1



The answers to Mini-Quiz questions are located in Appendix A.

1. Which of the following will return the square root of the number 16?
   a. `pow(16.0, 2)`
   b. `sqrt(16.0)`
   c. `sqrt(16.0, .5)`
   d. both a and b

2. Which of the following expressions will generate a random integer from 25 through 50, inclusive?
   a. `1 + rand() % (50 - 25 + 1)`
   b. `50 + rand() % (50 - 25 + 1)`
   c. `25 + rand() % (50 - 25 + 1)`
   d. `25 + rand() % (50 - 25 - 1)`

3. Which of the following functions initializes the random number generator in C++?
   a. `initialize()`
   b. `startRand()`
   c. `rand(time(0))`
   d. none of the above

4. Which of the following directives is necessary for a program to use the C++ `time` function?
   a. `#include <ctime>`
   b. `#include <stime>`
   c. `#include <time>`
   d. none of the above

Creating Program-Defined Value-Returning Functions

In addition to using the value-returning functions built into the C++ language, you also can create your own value-returning functions. As mentioned earlier, such functions are referred to as program-defined value-returning functions, because the function definitions are contained in the program itself rather than in a different file. You already know how to create one program-defined value-returning function: `main`. In this section, you will learn how to create other program-defined value-returning functions. The first one you will create is for the random addition problems program from the previous section. Because that program is short and simple, it's perfectly acceptable to have the `main` function perform all of the program's tasks, as it does now. However, when coding large and complex programs, programmers typically divide the program into small and manageable tasks and then assign some of the tasks to program-defined functions. Assigning tasks to program-defined functions makes the program easier to code, because it allows the programmer to concentrate on coding one small piece of the program at a time. It also allows more than one programmer to work on a program at the same time. In this case, you will remove the task of generating the random numbers from the `main` function and assign that responsibility to a program-defined value-returning function instead. Figure 9-13 shows the modified IPO chart for the `main` function. The changes made to the function's original IPO chart, shown earlier in Figure 9-6, are shaded in Figure 9-13. The figure also shows the IPO chart for a program-defined value-returning function named `getRandomNumber`. When invoked by a statement in the `main` function, the `getRandomNumber` function will generate a random number from one through 10 and then return the random number to the statement that called it. The `main` function will need to call the `getRandomNumber` function twice, because the program needs two random numbers and the `getRandomNumber` function can return only one random number at a time.



Recall that a value-returning function can return only one value at a time.

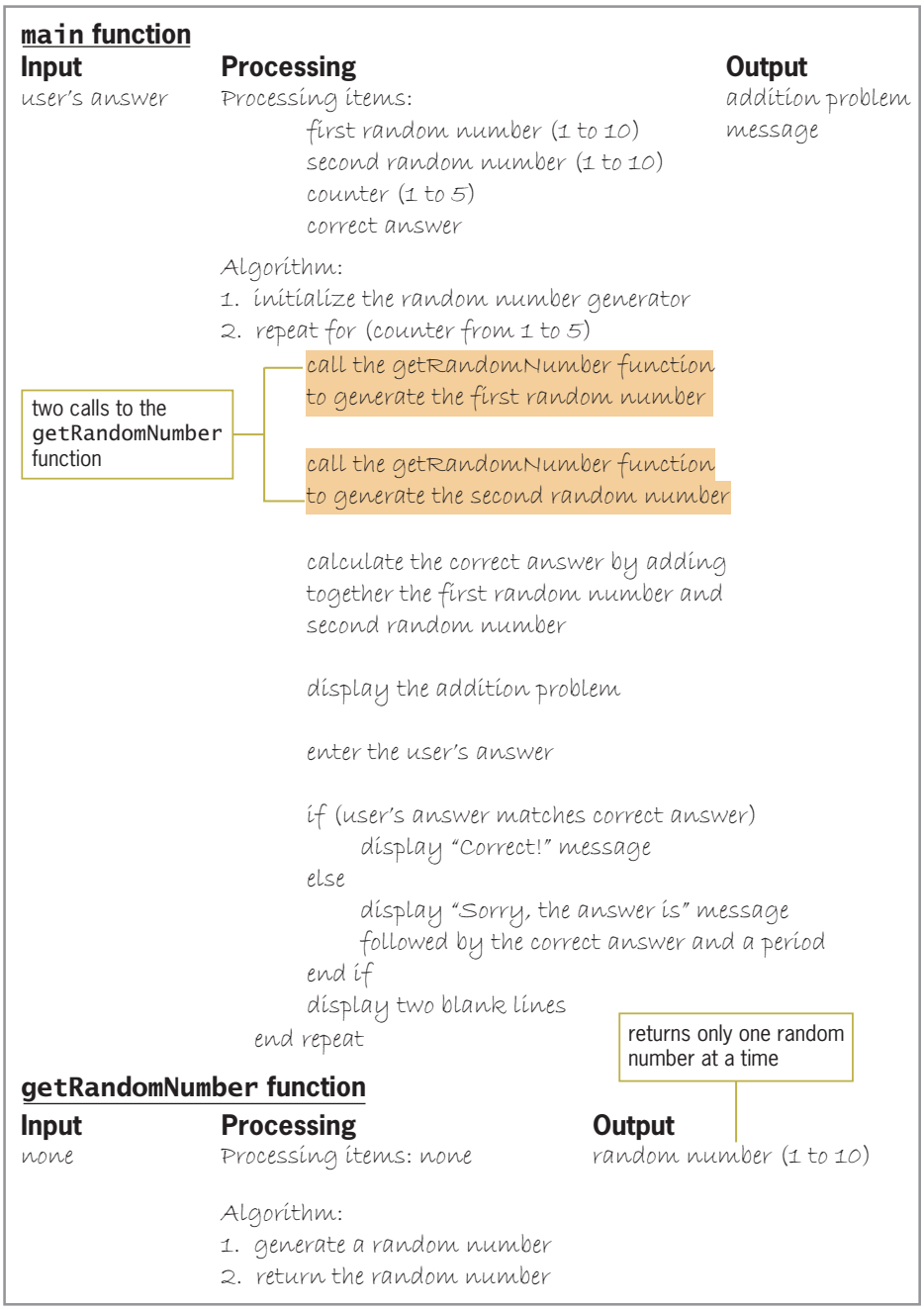

main function
Input
user's answer
Processing
Processing items:
first random number (1 to 10)
second random number (1 to 10)
counter (1 to 5)
correct answer
Output
addition problem
message
Algorithm:
1. initialize the random number generator
2. repeat for (counter from 1 to 5)
call the getRandomNumber function to generate the first random number
call the getRandomNumber function to generate the second random number
calculate the correct answer by adding together the first random number and second random number
display the addition problem
enter the user's answer
if (user's answer matches correct answer)
display "Correct!" message
else
display "Sorry, the answer is" message followed by the correct answer and a period
end if
display two blank lines
end repeat
two calls to the getRandomNumber function
returns only one random number at a time
getRandomNumber function
Input
none
Processing
Processing items: none
Output
random number (1 to 10)
Algorithm:
1. generate a random number
2. return the random number

Figure 9-13 IPO charts for the modified random addition problems program

Figure 9-14 shows the syntax used to create (or define) a value-returning function in a C++ program. The figure also shows examples of program-defined value-returning functions. The `getRandomNumber` function in Example 1 returns a random integer from one through 10. The `getRectangleArea` function in Example 2 calculates and returns the area of a rectangle, using the values stored in the `len` and `wid` variables. The values for the `len` and `wid` variables will be passed to the function by the statement from which it is called. The function returns the area as a `double` number. The `getBonus` function in Example 3 uses the values passed to it to calculate the amount of a salesperson's bonus, which the function returns as a `double` number. For now, don't be concerned if you don't fully understand the examples. They will become clearer to you as you progress through the chapter.



You can define a default value for one or more of a function's formal parameters. This topic is covered in Computer Exercise 25.

HOW TO Create a Program-Defined Value-Returning Function

Syntax

```
returnDataType functionName([parameterList])        function header
{
        one or more statements                      function body
        return expression;
}       //end of functionName function
```

Example 1

```
int getRandomNumber()                               function definition
{
      int randInteger = 0;
      randInteger = 1 + rand() % (10 - 1 + 1);
      return randInteger;
}     //end of getRandomNumber function
```

The function generates a random integer from one through 10 and then returns the random integer.

Example 2

```
double getRectangleArea(double len, double wid)
{
      return len * wid;
}     //end of getRectangleArea function
```

The function calculates the area of a rectangle and then returns the result as a `double` number.

Example 3

```
double getBonus(int sold, double bonusRate)
{
      double bonus = 0.0;

      bonus = sold * bonusRate;
      return bonus;
}     //end of getBonus function
```

The function calculates the amount of a salesperson's bonus and then returns the result as a `double` number.

Figure 9-14 How to create a program-defined value-returning function



The most commonly used data types are listed in Figure 3-5 in Chapter 3. The rules for naming variables are listed in Figure 3-2 in Chapter 3.

As Figure 9-14 indicates, a function definition is composed of a function header and a function body. The function header (the first line in a function definition) for a value-returning function begins with *returnDataType*, which indicates the data type of the value the function returns. The `getRandomNumber` function in Figure 9-14 returns an integer; therefore, its *returnDataType* is `int`. The `getRectangleArea` and `getBonus` functions, on the other hand, return a `double` number and have a *returnDataType* of `double`. The function header also specifies the name of the function. The rules for naming functions are the same as for naming variables. However, it is a common practice to begin a function's name with a verb. To make your programs more self-documenting and easier to understand, you should use meaningful names that describe the task the function performs. In the examples in Figure 9-14, the names `getRandomNumber`, `getRectangleArea`, and `getBonus` indicate that the functions return a random number, the

area of a rectangle, and a bonus amount, respectively. The function header also contains an optional *parameterList* enclosed in parentheses. Keep in mind that only the *parameterList* is optional; the parentheses are a required part of the syntax. The *parameterList* contains the data type and name of one or more memory locations. The memory locations in a function's *parameterList* are called **formal parameters**. Each formal parameter will store an item of information that is passed to the function when the function is called. In Example 1 in Figure 9-14, the empty set of parentheses in the function header indicates that the `getRandomNumber` function will not be passed any information by the statement that calls it. The function header in Example 2, however, contains two formal parameters and indicates that the `getRectangleArea` function will be passed two items of information when it is invoked. Both items will have the `double` data type. The `getBonus` function in Example 3 will receive an `int` item followed by a `double` item from the statement that invokes it. You will learn more about the *parameterList* later in the chapter and also in Chapter 10.



Rather than using an empty set of parentheses when a function is not passed any information, some programmers enter the keyword `void` within the parentheses.

In addition to the function header, a function definition also contains a function body. The function body contains the instructions for performing the function's assigned task. The function body begins with the opening brace (`{`) and ends with the closing brace (`}`). In most cases, the last statement in the function body of a value-returning function is `return` *expression*`;`, in which *expression* represents the function's one and only return value. The data type of the *expression* must agree with the *returnDataType* specified in the function header. The **return statement** returns the *expression*'s value to the statement that called the function. After the `return` statement is processed, the function ends and program execution continues in the calling function. Although not a requirement, it is a good programming practice to use a comment (such as `//end of getRandomNumber function`) to mark the end of a program-defined function. The comment will make your program easier to read and understand.

Mini-Quiz 9-2



The answers to Mini-Quiz questions are located in Appendix A.

1. A function header contains _____.
   a. the data type of the function's return value
   b. the function's name
   c. an optional *parameterList*
   d. all of the above

2. Which of the following is a valid function header for the `getArea` function? The function returns a `double` number and does not have any formal parameters.
   a. `double getArea()`
   b. `double getArea`
   c. `double getArea();`
   d. `double getArea;`

3. Write the function header for the `getGrossPay` function. The function returns a `double` number and has two formal parameters: an `int` variable named `hours` and a `double` variable named `rate`.

4. The `getGrossPay` function from Question 3 calculates and returns an employee's gross pay. Write a C++ statement that returns the gross pay to the statement that called the function. The gross pay is stored in a `double` variable named `gross`.

Calling a Function

For a function to perform its task, it must be called (or invoked). The `main` function in a C++ program is invoked automatically when the program is executed. Functions other than `main`, however, must be called by a statement within the program. You do this by including the function's name and actual arguments (if any) in the statement, as indicated in the syntax and examples in Figure 9-15. The actual arguments are listed in the optional *argumentList*, which appears within parentheses after the function's name. Value-returning functions typically are called from statements that do one of the following: assign the return value to a variable, use the return value in a calculation or comparison, or display the return value. The `cout` statement in Example 1, for instance, calls the built-in value-returning `rand` function and then displays the function's return value on the computer screen. The assignment statement in Example 2 invokes the built-in value-returning `sqrt` function, passing it one actual argument: the `double` number 100.0. The `sqrt` function calculates the square root of the number 100.0 and then returns the answer (10.0) to the assignment statement, which assigns the answer to the `squareRoot` variable. Unlike a call to a value-returning function, a call to a void function is an independent statement. In other words, it's not part of a statement that either assigns or displays the return value, because a void function does not return a value. Example 3 shows a statement that calls a built-in void function: `srand`. Recall that you learned about the `srand` function earlier in the chapter. The `srand(5);` statement passes the integer 5 to the function, which uses the integer to initialize the random number generator. As mentioned earlier, you will learn more about void functions in Chapter 10. Although the examples just cited call built-in functions, the same method is used to call program-defined functions. The assignment statement in Example 4, for instance, calls the program-defined value-returning `getRandomNumber` function shown earlier in Figure 9-14. The function generates a random integer from one through 10 and then returns the integer to the assignment statement, which assigns it to the `num1` variable. The function calls in the remaining two examples in Figure 9-15 pass actual arguments to program-defined functions. An actual argument can be a variable, named constant, literal constant, or keyword; however, in most cases it will be a variable. Each variable you declare in a program has both a value and a unique address that represents the location of the variable in the computer's internal memory. C++ allows you to pass either the variable's value or its

address to a function. Passing a variable's value is referred to as **passing *by value***. Passing a variable's address is referred to as **passing *by reference***. Unless you specify otherwise, variables in C++ are automatically passed *by value*. For now, you do not need to concern yourself with passing *by reference*, because all variables passed to functions in this chapter are passed *by value*. You will learn how to pass variables *by reference* in Chapter 10. The number of actual arguments passed to a function should match the number of formal parameters in its function header. In addition, the data type and position of each actual argument must agree with the data type and position of its corresponding formal parameter. This is because, when the function is called, the computer stores the value of the first actual argument in the function's first formal parameter, the value of the second actual argument in its second formal parameter, and so on. The function call in Example 5, for instance, passes two `double` numbers to the `getRectangleArea` function. This is because the function's header (shown earlier in Figure 9-14) contains two formal parameters, both of which have the `double` data type. The function uses the values stored in its formal parameters to calculate the area. It then returns the area to the statement that called it. In this case, it returns the area to the `cout << getRectangleArea(7.25, 21.0);` statement, which displays the area on the computer screen. The `if` clause in Example 6 invokes the `getBonus` function. The function's header (shown earlier in Figure 9-14) indicates that the function is expecting to receive two values, in this order: an integer that represents the amount sold and a `double` number that represents the bonus rate. Because of this, the function call in Example 6 passes two actual arguments in the required data type and order: the `int sales` variable first and the `double rate` variable second. The computer stores the values of the actual arguments in the `getBonus` function's formal parameters, which are named `sold` and `bonusRate`. Notice that the names of the actual arguments do not have to be identical to the names of their corresponding formal parameters. In fact, to avoid confusion, it usually is better to use different names for the actual arguments and formal parameters. The `getBonus` function uses the values in its formal parameters to calculate the salesperson's bonus. It then returns the bonus as a `double` number to the `if` clause, whose condition compares the return value to the `double` number 999.99. The condition will evaluate to true when the return value is greater than 999.99. It will evaluate to false when the return value is not greater than 999.99. In other words, it will evaluate to false when the return value is either less than or equal to 999.99. Keep in mind that when the computer encounters a statement that calls a function, it temporarily leaves the calling function to process the code contained in the called function. It returns to the calling function only after the called function ends.



You can define a default value for one or more of a function's formal parameters. This topic is covered in Computer Exercise 25.

HOW TO Call a Function

Syntax

functionName(**[***argumentList***]**)

Example 1

```
cout << rand();
```

The `cout` statement calls the built-in value-returning `rand` function and then displays the function's return value on the computer screen.

Example 2

```
double squareRoot = 0.0;
squareRoot = sqrt(100.0);
```

The assignment statement calls the built-in value-returning `sqrt` function, passing it the `double` number 100.0. It then displays the function's return value on the computer screen.

Example 3

```
srand(5);
```

a void function call is a self-contained statement

The statement calls the built-in void `srand` function, passing it the integer 5. The function uses the integer to initialize the random number generator.

Example 4

```
int num1 = 0;
num1 = getRandomNumber();
```

The assignment statement calls the `getRandomNumber` function and then assigns the function's return value to the `num1` variable.

Example 5

```
cout << getRectangleArea(7.25, 21.0);
```

The `cout` statement calls the `getRectangleArea` function, passing it the `double` numbers 7.25 and 21.0. It then displays the function's return value on the computer screen.

Example 6

```
int sales = 0;
double rate = 0.0;
cin >> sales;
cin >> rate;
if (getBonus(sales, rate) > 999.99)
```

The `if` clause calls the `getBonus` function, passing it the integer stored in the `sales` variable and the `double` number stored in the `rate` variable. It then compares the function's return value to the `double` number 999.99.

Figure 9-15 How to call a function

Now that you know how to both create and call a program-defined value-returning function, you can code the `main` and `getRandomNumber` functions. Figure 9-16 shows the IPO chart information and C++ instructions for both functions. The statements that call the `getRandomNumber` function are shaded in the figure.

main function

IPO chart information	C++ instructions
Input	
user's answer	`int userAnswer = 0;`
Processing	
first random number (1 to 10)	`int num1 = 0;`
second random number (1 to 10)	`int num2 = 0;`
counter (1 to 5)	this variable is created and initialized in the for clause
correct answer	`int correctAnswer = 0;`
Output	
addition problem	this contains string literal constants and the num1 and num2 variables
message	this is one of two messages composed of either a string literal constant or string literal constants and the correctAnswer variable
Algorithm	
1. initialize the random number generator	`srand(static_cast<int>(time(0)));`
2. repeat for (counter from 1 to 5)	`for (int x = 1; x < 6; x += 1)`
	`{`
call the getRandomNumber function to generate the first random number	`num1 = getRandomNumber();`
call the getRandomNumber function to generate the second random number	`num2 = getRandomNumber();`
calculate the correct answer by adding together the first random number and second random number	`correctAnswer = num1 + num2;`
display the addition problem	`cout << "What is the sum of " << num1 << " + " << num2 << "? ";`
enter the user's answer	`cin >> userAnswer;`
if (user's answer matches correct answer)	`if (userAnswer == correctAnswer)`
display "Correct!" message	`cout << "Correct!";`
else	`else`
display "Sorry, the answer is" message followed by the correct answer and a period	`cout << "Sorry, the correct answer is " << correctAnswer << ".";`
end if	`//end if`
display two blank lines	`cout << endl << endl;`
end repeat	`} //end for`

Figure 9-16 IPO chart information and C++ instructions for the modified random addition problems program *(continues)*



The IPO charts for the `main` and `getRandomNumber` functions are shown earlier in Figure 9-13.

(continued)

`getRandomNumber` function

IPO chart information	**C++ instructions**
Input	
none	
Processing	
none	
Output	
random number (1 to 10)	`int randInteger = 0;`
Algorithm	
1. generate a random number	`randInteger = 1 + rand()` `% (10 - 1 + 1);`
2. return the random number	`return randInteger;`

Figure 9-16 IPO chart information and C++ instructions for the modified random addition problems program

Function Prototypes

Most C++ programmers enter the function definitions below the `main` function in a program. When a function definition appears below the `main` function, you must enter a function prototype *above* the `main` function; otherwise, the compiler won't recognize the function's name when it is used in the `main` function. A **function prototype** is a statement that specifies the function's name, the data type of its return value, and the data type of each of its formal parameters (if any). You also can include the name of each of the formal parameters; however, that is not a requirement. A program will have one function prototype for each function defined below the `main` function. You usually place the function prototypes at the beginning of the program, after the `#include` directives and `using namespace std;` statement. A function prototype alerts the C++ compiler that the function will be defined later in the program. The function prototypes in a program are similar to the table of contents in a book. As does each entry in a table of contents, each prototype is simply a preview of what will be expanded on later in the program (or in the book). Keep in mind that a function prototype is necessary only when the function is defined *below* the `main` function in the program. It is not needed for a function whose definition appears *above* the `main` function. In this book, the function definitions will be entered below the `main` function, because that is the format used by most C++ programmers. This means that each program-defined function will need a corresponding function prototype above the `main` function.

Figure 9-17 shows a function prototype's syntax, which is almost identical to a function header's syntax. However, unlike a function header, a function prototype ends with a semicolon. Also included in Figure 9-17 are function prototypes for the functions defined earlier in Figure 9-14. As Examples 2 and 3 indicate, it is not necessary to include the names of the formal parameters in a function prototype. However, many programmers include the names to make the program easier to read and understand. Some also include the

names for convenience, because it makes entering the function prototype an easy task. All you need to do is copy the function's header, then paste it in the function prototype section of the program, and then type a semicolon at the end of it.

HOW TO Write a Function Prototype

Syntax

returnDataType functionName(**[***parameterList***]**);

semicolon

each formal parameter's data type and (optionally) name

Example 1

```
int getRandomNumber();
```

Example 2

```
double getRectangleArea(double len, double wid);
```

or

only the data type of each formal parameter is required

```
double getRectangleArea(double, double);
```

Example 3

```
double getBonus(int sold, double bonusRate);
```

or

```
double getBonus(int, double);
```

Figure 9-17 How to write a function prototype

Figure 9-18 shows the complete code for the modified random numbers addition program. Changes made to the original program's code (shown earlier in Figure 9-11) are shaded in the figure. The program contains a function prototype on Line 13. The prototype alerts the computer that the `getRandomNumber` function is defined somewhere below the `main` function in the program. As indicated in the figure, the function definition appears on Lines 55 through 61. Some programmers use a comment (such as `//*****function definitions*****`) to separate the function definitions from the `main` function, as shown on Line 54; however, this is not a requirement. The statements that call the `getRandomNumber` function are on Lines 31 and 32. When the computer processes the `num1 = getRandomNumber();` statement, it temporarily leaves the `main` function to process the `getRandomNumber` function's code. The function header is processed first and simply marks the beginning of the function. The statements in the function body are processed next. The first statement creates and initializes the `randInteger` variable. The second statement generates a random integer from one through 10 and then assigns the random integer to the variable. The third (and last) statement returns the value stored in the `randInteger` variable to the statement that called the function. In this case, it returns the value to the `num1 = getRandomNumber();` statement, which assigns the value to the `num1` variable. After the `return` statement

is processed, the `getRandomNumber` function ends and the computer removes the `randInteger` variable from its internal memory. Processing continues with the statement that called the `getRandomNumber` function. The same procedure is followed when the computer processes the `num2 = getRandomNumber();` statement, except the `getRandomNumber` function's return value is assigned to the `num2` variable.

```
 1  //Modified Random Addition.cpp
 2  //Displays random addition problems
 3  //Allows the user to enter the answer and then
 4  //displays a message that indicates whether the
 5  //user's answer is correct or incorrect
 6  //Created/revised by <your name> on <current date>
 7
 8  #include <iostream>
 9  #include <ctime>
10  using namespace std;
11
12  //function prototype
13  int getRandomNumber();
14
15  int main()
16  {
17     //declare variables
18     int num1          = 0;
19     int num2          = 0;
20     int correctAnswer = 0;
21     int userAnswer    = 0;
22
23     //initialize rand function
24     srand(static_cast<int>(time(0)));
25
26     for (int x = 1; x < 6; x += 1)
27     {
28          //generate two random integers
29          //from 1 through 10, then
30          //calculate the sum
31          num1 = getRandomNumber();
32          num2 = getRandomNumber();
33          correctAnswer = num1 + num2;
34
35          //display addition problem and get user's answer
36          cout << "What is the sum of " << num1
37               << " + " << num2 << "? ";
38          cin >> userAnswer;
39
40          //detemine whether user's answer is correct
41          if (userAnswer == correctAnswer)
42               cout << "Correct!";
```

function prototype

function calls

Figure 9-18 Modified random addition problems program *(continues)*

(continued)

```
43          else
44               cout << "Sorry, the correct answer is "
45                    << correctAnswer << ".";
46          //end if
47          cout << endl << endl;
48     }    //end for
49
50     system("pause");
51     return 0;
52  }  //end of main function
53
54  //*****function definitions*****
55  int getRandomNumber()
56  {
57     int randInteger = 0;
58     //generate random integer from 1 through 10
59     randInteger = 1 + rand() % (10 - 1 + 1);
60     return randInteger;
61  }  //end of getRandomNumber function
```

your C++ development tool may not require this statement

function definition

Figure 9-18 Modified random addition problems program

The Plano Elementary School Program

The principal of Plano Elementary School would like to use the random addition problems program that you completed in the previous section. However, she wants the ability to specify the range of random numbers that appear each time the program is run. This will require you to make a few modifications to the program shown in Figure 9-18. More specifically, you will modify the program to allow the user to enter the smallest and largest integer in the desired range. You then will have the two function calls in the program send that information to the `getRandomNumber` function. Figure 9-19 shows the Plano Elementary School program. The changes made to the code shown in Figure 9-18 are shaded in Figure 9-19. The Plano Elementary School program provides an example of code reuse. It also demonstrates how you can use functions to improve programming productivity by splitting larger problems into a series of smaller problems. Each small problem can be assigned to a member of the programming team. In this case, for example, you can code the `main` function while your friend Sam codes the `getRandomNumber` function. By splitting the work in this manner, you won't need to worry about the code for generating the random numbers. Whenever the `main` function needs a random number, you can simply call Sam's function to get one. Actually, you can use Sam's function in any program that requires a random integer within a specific range; you just need to enter the function definition in the program. (Depending on where you enter the function definition, you also may need the function prototype.) Sam also will be using another programmer's code in his `getRandomNumber` function. More specifically, he'll be using the C++ built-in `rand` function. Figure 9-20 shows a sample run of the Plano Elementary School program.

```
1  //Plano Elementary.cpp
2  //Displays random addition problems
3  //Allows the user to enter the answer and then
4  //displays a message that indicates whether the
5  //user's answer is correct or incorrect
6  //Created/revised by <your name> on <current date>
7
8  #include <iostream>
9  #include <ctime>
10 using namespace std;
11
12 //function prototype
13 int getRandomNumber(int lower, int upper);
14
15 int main()
16 {
17    //declare variables
18    int smallest      = 0;
19    int largest       = 0;
20    int num1          = 0;
21    int num2          = 0;
22    int correctAnswer = 0;
23    int userAnswer    = 0;
24
25    //initialize rand function
26    srand(static_cast<int>(time(0)));
27
28    cout << "Smallest integer: ";
29    cin >> smallest;
30    cout << "Largest integer: ";
31    cin >> largest;
32    cout << endl;
33
34    for (int x = 1; x < 6; x += 1)
35    {
36        //generate two random integers
37        //from smallest through largest, then
38        //calculate the sum
39        num1 = getRandomNumber(smallest, largest);
40        num2 = getRandomNumber(smallest, largest);
41        correctAnswer = num1 + num2;
42
43        //display addition problem and get user's answer
44        cout << "What is the sum of " << num1
45             << " + " << num2 << "? ";
46        cin >> userAnswer;
47
48        //detemine whether user's answer is correct
49        if (userAnswer == correctAnswer)
50             cout << "Correct!";
51        else
52            cout << "Sorry, the correct answer is "
53                 << correctAnswer << ".";
54        //end if
```

the names are not required

gets the smallest and largest integers in the range

passes the smallest and largest integers to the `getRandomNumber` function

Figure 9-19 Plano Elementary School program *(continues)*

(continued)

```
55          cout << endl << endl;
56     }    //end for
57
58     system("pause");
59     return 0;
60  }  //end of main function
61
62  //*****function definitions*****
63  int getRandomNumber(int lower, int upper)
64  {
65     int randInteger = 0;
66     //generate random integer from lower through upper
67     randInteger = lower + rand() % (upper - lower + 1);
68     return randInteger;
69  }  //end of getRandomNumber function
```

your C++ development tool may not require this statement

receives the smallest and largest integers from each function call on Lines 39 and 40

Figure 9-19 Plano Elementary School program

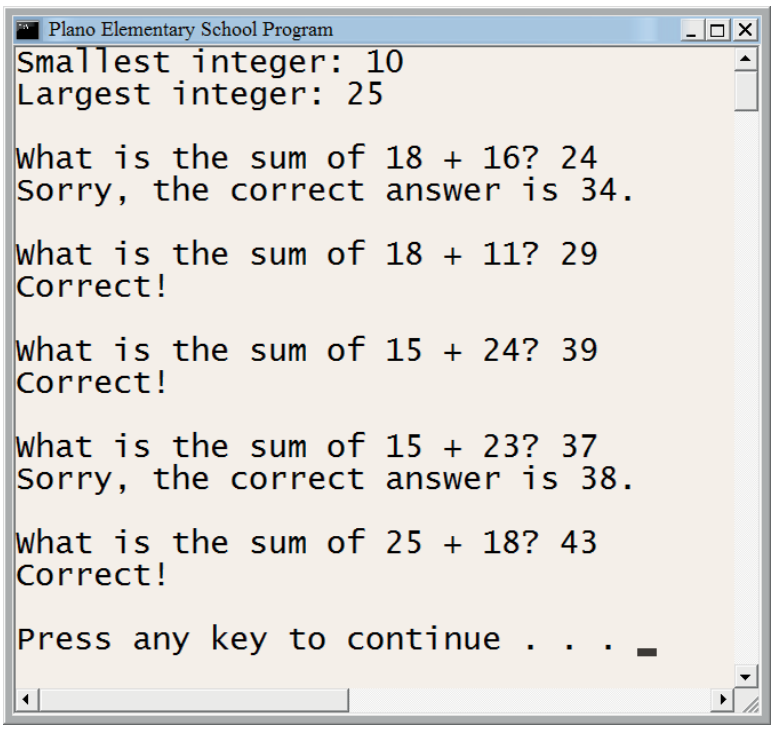

Plano Elementary School Program
Smallest integer: 10
Largest integer: 25
What is the sum of 18 + 16? 24
Sorry, the correct answer is 34.
What is the sum of 18 + 11? 29
Correct!
What is the sum of 15 + 24? 39
Correct!
What is the sum of 15 + 23? 37
Sorry, the correct answer is 38.
What is the sum of 25 + 18? 43
Correct!
Press any key to continue . . .

Figure 9-20 Sample run of the Plano Elementary School program



The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 9-3

1. The `getArea` function returns a `double` number and has no formal parameters. Which of the following calls the `getArea` function and assigns its return value to a `double` variable named `area`?

   a. `area = getArea`

   b. `area = getArea();`

   c. `area = getArea(double);`

   d. `getArea(area);`

2. Which of the following is a valid function prototype for the `getArea` function from Question 1?

   a. `double getArea()`

   b. `double getArea`

   c. `double getArea();`

   d. `double getArea;`

3. Write a C++ statement that will display the value returned by the `getArea` function from Question 1.

4. Write a function prototype for the `getGrossPay` function. The function returns a `double` number and has two formal parameters: an `int` variable named `hours` and a `double` variable named `rate`.

5. Write a statement that invokes the `getGrossPay` function from Question 4. The statement should pass the function the integer 40 and the value stored in a `double` variable named `payRate`. The statement should assign the function's return value to a `double` variable named `weekGross`.

The Area Calculator Program

Figure 9-21 shows the problem specification and IPO chart for the area calculator program, and Figure 9-22 shows the program's code. (The flowchart for this program is contained in the Ch9Flowcharts.pdf file, which is located in the Cpp6\Chap09 folder.) The program uses a program-defined value-returning function to calculate the area of a rectangle, given the rectangle's length and width measurements. The program displays the area on the computer screen. Figure 9-23 shows a sample run of the program.

Problem specification
Create a program that allows the user to enter a rectangle's length and width (in feet). The program should calculate and display the rectangle's area in square feet.

`main` function

Input	Processing	Output
length (feet) width (feet)	Processing items: none Algorithm: 1. enter the length and width 2. call the getRectangleArea function to calculate the area; pass the length and width 3. display the area	area (square feet)

`getRectangleArea` function

Input	Processing	Output
length (feet) width (feet)	Processing items: none Algorithm: 1. return the area, which is the result of multiplying the length by the width	area (square feet)

Figure 9-21 Problem specification and IPO charts for the area calculator program

```
 1  //Area Calculator.cpp - displays the area of a rectangle
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  using namespace std;
 6
 7  //function prototype
 8  double getRectangleArea(double len, double wid);
 9
10  int main()
11  {
12     double length = 0.0;
13     double width  = 0.0;
14     double area   = 0.0;
15
16     cout << "Rectangle length (in feet): ";
17     cin >> length;
18     cout << "Rectangle width (in feet): ";
19     cin >> width;
20
21     area = getRectangleArea(length, width);
22     cout << "Area: " << area << " square feet" << endl;
23
24     system("pause");
25     return 0;
26  }  //end of main function
27
28  //*****function definitions*****
29  double getRectangleArea(double len, double wid)
30  {
31     return len * wid;
32  }  //end of getRectangleArea function
```

the names are not required

function call

your C++ development tool may not require this statement

function header

Figure 9-22 Area calculator program

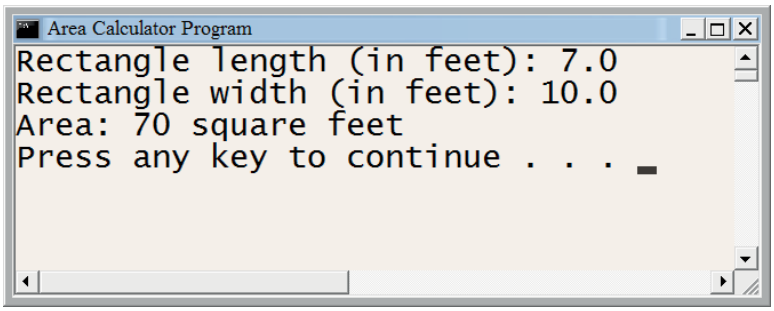

Area Calculator Program
Rectangle length (in feet): 7.0
Rectangle width (in feet): 10.0
Area: 70 square feet
Press any key to continue . . .

Figure 9-23 Sample run of the area calculator program

You will desk-check the area calculator program in Figure 9-22 using the `double` numbers 7.0 and 10.0 as the rectangle's length and width, respectively. Desk-checking the program will help you understand how the computer processes a program-defined value-returning function when it is invoked. The statements on Lines 12 through 14 in the program create and initialize three `double` variables named `length`, `width`, and `area`. The statements on Lines 16 through 19 prompt the user to enter the length and width of the rectangle and then store the user's responses in the `length` and `width` variables, respectively. Figure 9-24 shows the desk-check table after the statements are processed.



Figure 9-24 Desk-check table after the statements on Lines 12 through 19 are processed

The assignment statement on Line 21 calls the `getRectangleArea` function, passing it two actual arguments: the `double length` variable and the `double width` variable. Recall that unless specified otherwise, variables in C++ are passed *by value*, which means that only the contents of the variables are passed to the function. In this case, the computer passes the numbers 7.0 and 10.0 to the `getRectangleArea` function. At this point, the computer temporarily leaves the `main` function to process the `getRectangleArea` function's code, beginning with the function header on Line 29 in the program. The function header contains two formal parameters. The formal parameters tell the computer to reserve two memory locations: a `double` variable named `len` and a `double` variable named `wid`. After reserving the `len` and `wid` variables, the computer stores the values passed to the function—in this case, the numbers 7.0 and 10.0—in the variables. Figure 9-25 shows the desk-check table after the `getRectangleArea` function header is processed.

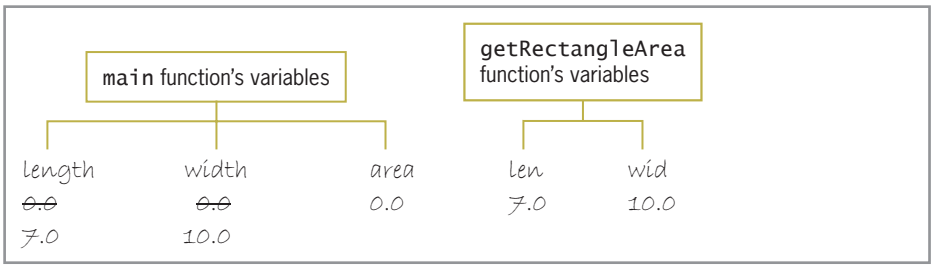


Figure 9-25 Desk-check table after the `getRectangleArea` function header is processed

Next, the computer processes the `return len * wid;` statement on Line 31 in the `getRectangleArea` function. The statement multiplies the `len` variable's value by the `wid` variable's value and then returns the result to the statement that called the `getRectangleArea` function. In this case, it returns the `double` number 70.0 to the assignment statement on Line 21 in the `main` function. The statement assigns the `double` number to the `area` variable. After the `getRectangleArea` function's `return` statement is processed, the function ends and the computer removes the `len` and `wid` variables from its internal memory. Figure 9-26 shows the desk-check table at this point in the program. Notice that only the `main` function's variables are still in the computer's internal memory.

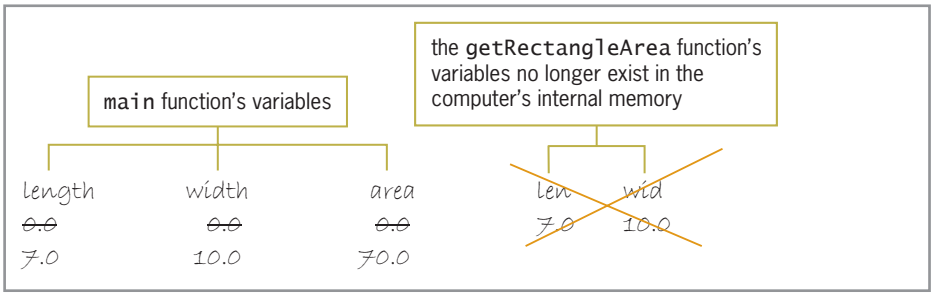


Figure 9-26 Desk-check table after the `getRectangleArea` function ends

Next, the computer processes the `cout` statement on Line 22. The statement displays the contents of the `area` variable on the screen. The computer processes the `system("pause");` and `return 0;` statements next. The `system("pause");` statement pauses program execution, and the `return 0;` statement returns the number 0 to the operating system to indicate that the program ended normally. After the `return 0;` statement is processed, the `main` function ends and the computer removes the `length`, `width`, and `area` variables from its internal memory. At this point, you may be wondering why the program needs to pass the contents of the `length` and `width` variables to the `getRectangleArea` function. Why can't the function just use both variables in its `return` statement, like this: `return length * width;`? You also may be wondering why the computer removes the `len` and `wid` variables from memory after the `getRectangleArea` function ends, but waits until the `main` function ends before it removes the `length`, `width`, and `area` variables. Why are the variables removed from memory at different times? To answer these questions, you will need to learn about the scope and lifetime of a variable. The scope and lifetime of a variable are the last topics covered in this chapter.

The Scope and Lifetime of a Variable

A variable's **scope** indicates where in the program the variable can be used, and its **lifetime** indicates how long the variable remains in the computer's internal memory. Although variables can have either local or global scope, most of the variables used in a program will have local scope. This is because fewer unintentional errors occur in programs when the variables are declared using the minimum scope needed, which usually is local scope. A variable's scope and lifetime are determined by where you declare the variable in the program. Variables declared within a function, and those that appear in a function's *parameterList*, have a local scope and are referred to as local variables. **Local variables** can be used only by the function in which they are declared or in whose *parameterList* they appear. Local variables remain in the computer's internal memory until the function ends, which is when the computer encounters the function's closing brace. Unlike local variables, **global variables** are declared outside of any function in the program, and they remain in memory until the program ends. Also unlike a local variable, any statement in the program can use a global variable. Declaring a variable as global rather than local allows unintentional errors to occur when a function that should not have access to the variable inadvertently changes the variable's contents. Because of this, you should avoid using global variables in your programs. If more than one function needs access to the same variable, it is better to create a local variable in one of the functions and then pass that variable to the other functions that need it.



You can experiment with the concepts of scope and lifetime by completing Computer Exercise 19 at the end of the chapter.

In the area calculator program shown earlier in Figure 9-22, the `length`, `width`, and `area` variables are declared on Lines 12 through 14 in the `main` function. As a result, the variables are local to the `main` function and can be used only by statements below Line 14 within the `main` function. The `getRectangleArea` function is not even aware of the existence of these variables in memory. If you want the `getRectangleArea` function to use the values stored in the `length` and `width` variables, you will need to pass each variable's value to the function. The `len` and `wid` variables, on the other hand, are local to the `getRectangleArea` function because they appear in the function's *parameterList*. Therefore, only the statements within the `getRectangleArea` function can use the `len` and `wid` variables. As mentioned earlier, local variables remain in the computer's internal memory until the function in which they are created ends. This explains why the `len` and `wid` variables are removed from memory after the `getRectangleArea` function ends. It also explains why the computer waits until the `main` function ends before it removes the `length`, `width`, and `area` variables from memory. Now that you understand the concepts of scope and lifetime, you will view and desk-check a program that uses two program-defined value-returning functions.

The Bonus Calculator Program

Figure 9-27 shows the problem specification for the bonus calculator program. The program calculates and displays a salesperson's bonus, which is 5% of his or her sales. Figure 9-27 also shows the IPO chart information and C++ instructions for the `main` function, which calls two program-defined value-returning functions named `getSales` and `getBonus`. The IPO charts and C++ instructions for these functions are shown in Figures 9-28 and 9-29.

Problem specification
Create a program that allows the user to enter the amount of a salesperson's sales. The program should calculate a 5% bonus and then display the bonus on the computer screen.

`main` function

IPO chart information	**C++ instructions**
Input	
sales	`int sales = 0;`
bonus rate (5%)	the function will pass the literal constant .05 to the getBonus function
Processing	
none	
Output	
bonus	`double bonus = 0.0;`
Algorithm	
1. call the getSales function to get the sales	`sales = getSales();`
2. call the getBonus function to calculate the bonus; pass the function the sales and the bonus rate of .05	`bonus = getBonus(sales, .05);`
3. display the bonus	`cout << "Bonus: $ " << bonus << endl;`

Figure 9-27 Problem specification, IPO chart information, and C++ code for the `main` function

`getSales` function

IPO chart information	**C++ instructions**
Input	
sales	`int salesAmt = 0;`
Processing	
none	
Output	
sales	
Algorithm	
1. enter the sales	`cout << "Enter sales: ";` `cin >> salesAmt;`
2. return the sales	`return salesAmt;`

Figure 9-28 IPO chart information and C++ code for the `getSales` function

(continued)

getBonus function	
IPO chart information	**C++ instructions**
Input	
sales (formal parameter)	`int sold`
bonus rate (formal parameter)	`double bonusRate`
Processing	
none	
Output	
bonus	`double bonusAmt = 0.0;`
Algorithm	
1. calculate the bonus by multiplying the sales by the bonus rate	`bonusAmt = sold * bonusRate;`
2. return the bonus	`return bonusAmt;`

Figure 9-29 IPO chart information and C++ code for the `getBonus` function

The C++ code for the entire program is shown in Figure 9-30. The function calls appear on Lines 19 and 20 in the `main` function and are shaded in the figure. The function definitions are located below the `main` function, on Lines 31 through 44. The function prototypes are located above the `main` function, on Lines 9 and 10. Figure 9-31 shows a sample run of the bonus calculator program.

```
 1  //Bonus Calculator.cpp - displays the amount of a bonus
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <iomanip>
 6  using namespace std;
 7
 8  //function prototypes
 9  int getSales();
10  double getBonus(int sold, double bonusRate);
11
12  int main()
13  {
14     int sales    = 0;
15     double bonus = 0.0;
16
17     //call functions to get the sales and
18     //calculate the bonus
19     sales = getSales();
20     bonus = getBonus(sales, .05);
21
22     //display the bonus
23     cout << fixed << setprecision(2);
```

Figure 9-30 Bonus calculator program *(continues)*

```
24      cout << "Bonus: $ " << bonus << endl;
25
26      system("pause");
27      return 0;
28  }  //end of main function
29
30  //*****function definitions*****
31  int getSales()
32  {
33     int salesAmt = 0;
34     cout << "Enter sales: ";
35     cin >> salesAmt;
36     return salesAmt;
37  }  //end of getSales function
38
39  double getBonus(int sold, double bonusRate)
40  {
41     double bonus = 0.0;
42     bonus = sold * bonusRate;
43     return bonus;
44  }  //end of getBonus function
```

your C++ development tool may not require this statement

Figure 9-30 Bonus calculator program

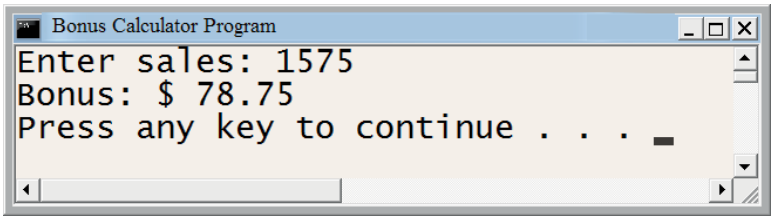


Figure 9-31 Sample run of the bonus calculator program

You will desk-check the program shown in Figure 9-30 using 1575 as the sales amount. The statements on Lines 14 and 15 create and initialize an `int` variable named `sales` and a `double` variable named `bonus`. Both variables are local to the `main` function, which means they can be used only within that function. Both variables will remain in the computer's internal memory until the `main` function ends. Figure 9-32 shows the desk-check table after the variable declaration statements on Lines 14 and 15 are processed.



Figure 9-32 Desk-check table after the variable declaration statements on Lines 14 and 15 are processed

When the computer encounters the `sales = getSales();` statement on Line 19, it temporarily leaves the `main` function to process the `getSales` function's code, beginning with the function header on Line 31. The function header does not contain any formal parameters, which indicates that the function will not receive any information when it is called. The first statement in the `getSales` function body creates and initializes an `int` variable named `salesAmt`. The variable is local to the `getSales` function, which means it can be used only within that function. The `salesAmt` variable will remain in the computer's internal memory until the `getSales` function ends. The next two statements in the `getSales` function prompt the user to enter a sales amount and then store the user's response—in this case, 1575—in the `salesAmt` variable. Figure 9-33 shows the sales amount entered in the desk-check table.

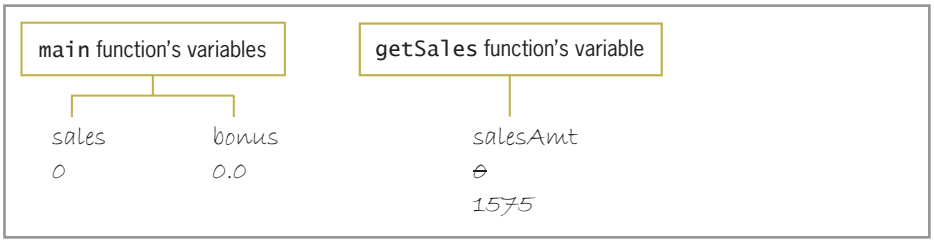


Figure 9-33 Desk-check table after the sales amount is entered

Next, the computer processes the `getSales` function's `return` statement, which returns the `salesAmt` variable's value to the statement that called the function. In this case, it returns the value to the assignment statement on Line 19 in the `main` function. The statement assigns the return value to the `sales` variable, as shown in Figure 9-34.

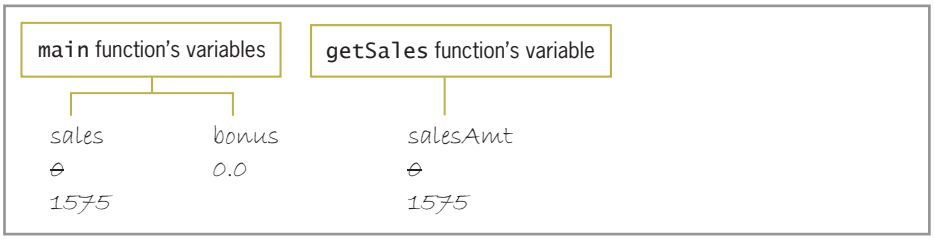


Figure 9-34 Desk-check table after the sales amount is returned to the `main` function

After its `return` statement is processed, the `getSales` function ends and the computer removes the `salesAmt` variable from memory. Figure 9-35 shows the desk-check table after the `getSales` function ends. Only the `main` function's variables are still in memory.

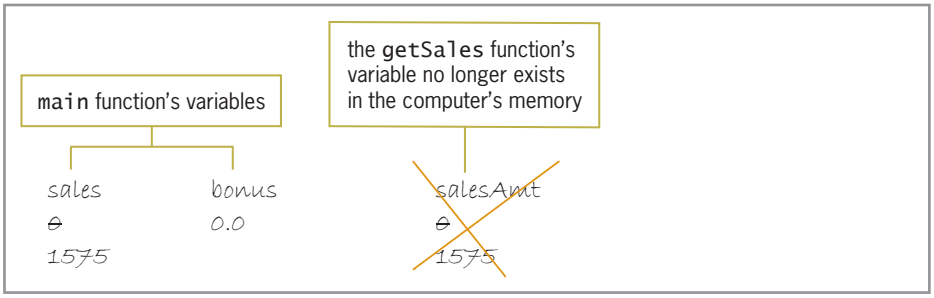


Figure 9-35 Desk-check table after the `getSales` function ends

Next, the computer processes the `bonus = getBonus(sales, .05);` statement on Line 20 in the `main` function. The statement calls the `getBonus` function, passing it two actual arguments. Here again, the computer temporarily leaves the `main` function. However, in this case, it does so to process the code in the `getBonus` function, beginning with the function header. The two formal parameters in the *parameterList* tell the computer to create two variables: an `int` variable named `sold` and a `double` variable named `bonusRate`. The variables are local to the `getBonus` function and can be used only within that function. The computer stores the first value passed to the function in the function's first formal parameter, and it stores the second value passed to the function in the function's second formal parameter. In this case, it stores the `sales` variable's value in the `sold` variable and the numeric literal constant .05 in the `bonusRate` variable, as shown in Figure 9-36.

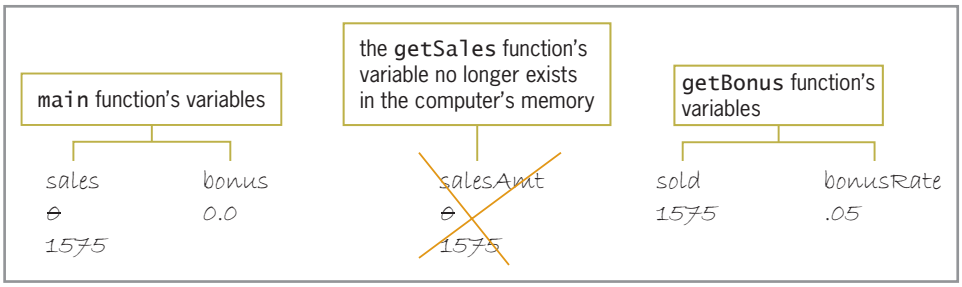


Figure 9-36 Desk-check table after the `getBonus` function header is processed

The first statement in the `getBonus` function declares an additional local variable: a `double` variable named `bonus`. The second statement multiplies the contents of the `sold` variable by the contents of the `bonusRate` variable and assigns the result (78.75) to the `bonus` variable, as shown in the desk-check table in Figure 9-37. The desk-check table indicates that two locations in the computer's memory have the same name: `bonus`. When the `bonus` name appears in a statement, the computer uses the position of the statement in the program to determine which of the two locations to use. If the program statement appears in the `main` function, the computer uses the `bonus` variable located in the `main` function's section in memory. However, if the program statement appears in the `getBonus` function, the computer uses the `bonus` variable located in the `getBonus` function's section in memory.

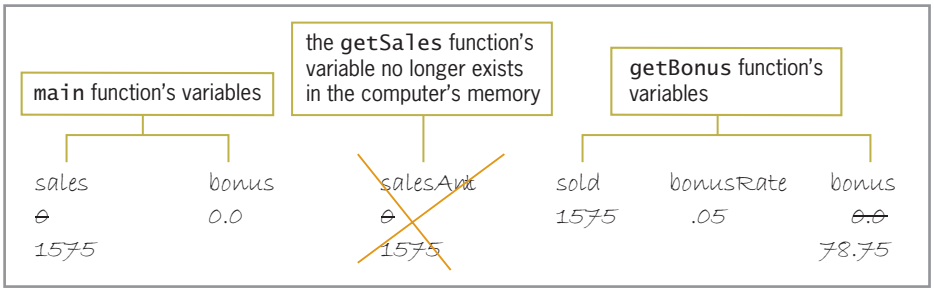


Figure 9-37 Desk-check table after the bonus is calculated

Next, the computer processes the `getBonus` function's `return` statement.

The statement returns the value stored in the function's `bonus` variable to the statement that called the function. In this case, it returns the value to the assignment statement on Line 20 in the `main` function. The statement assigns the value it receives to the `main` function's `bonus` variable. At this point, the `getBonus` function ends and the computer removes the function's local variables (`sold`, `bonusRate`, and `bonus`) from memory. Figure 9-38 shows the desk-check table after the `getBonus` function ends. Only the `main` function's variables are still in the computer's internal memory.

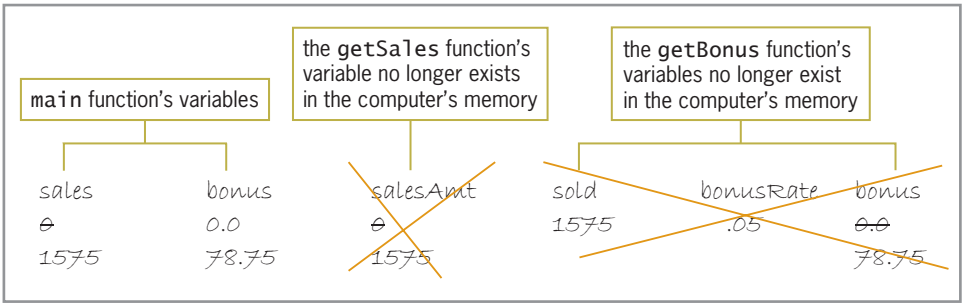


Figure 9-38 Desk-check table after the `getBonus` function ends

After the `getBonus` function ends, the computer processes the remaining instructions in the `main` function. The `cout` statements on Lines 23 and 24 display the bonus in fixed-point notation with two decimal places. The `system("pause");` statement on Line 26 pauses program execution, and the `return 0;` statement on Line 27 returns the number 0 to the operating system. After the `return 0;` statement is processed, the program ends and the computer removes the `main` function's variables (`sales` and `bonus`) from memory.



The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 9-4

1. A variable's lifetime indicates the portions of a program that can use the variable.
   a. True
   b. False

2. Unless specified otherwise, variables in C++ are passed *by contents*.
   a. True
   b. False

3. The variables in a function header have local scope.
   a. True
   b. False

4. Two functions in a program declare a variable using the same name. How does the computer know which variable to use?



LAB 9-1 Stop and Analyze

Study the program shown in Figure 9-39, and then answer the questions.



The answers to the labs are located in Appendix A.


```
 1  //Lab9-1.cpp - simulates a number guessing game
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <ctime>
 6  using namespace std;
 7
 8  int main()
 9  {
10     //declare variables
11     int randomNumber = 0;
12     int numberGuess  = 0;
13
14     //generate a random number from 1 through 10
15     srand(static_cast<int>(time(0)));
16     randomNumber = 1 + rand() % (10 - 1 + 1);
17
18     //get first number guess from user
19     cout << "Guess a number from 1 through 10: ";
20     cin >> numberGuess;
21
22     while (numberGuess != randomNumber)
23     {
24          cout << "Sorry, guess again: ";
25          cin >> numberGuess;
26     }    //end while
27
28     cout << endl << "Yes, the number is "
29           << randomNumber << "." << endl;
30
31     system("pause");
32     return 0;
33  }   //end of main function
```

your C++ development tool may not require this statement

Figure 9-39 Code for Lab 9-1

QUESTIONS

1. Why is the instruction on Line 5 necessary?
2. What is the purpose of the statement on Line 15?
3. If the `rand` function on Line 16 returns the number 453, what number will be assigned to the `randomNumber` variable?
4. Follow the instructions for starting C++ and opening the Lab9-1.cpp file. The file is contained in either the Cpp6\Chap09\Lab9-1 Project folder or the Cpp6\Chap09 folder. Run the program. Enter a number from 1 through 10. If you entered the correct number, the program displays the "Yes, the number is *x*." message, in which *x* is the number

you entered. If you did not enter the correct number, the program displays the "Sorry, guess again:" message. Continue entering numbers until you guess the correct number.

5. Modify the program so that it allows the user to make only four incorrect guesses. When the user has made the fourth incorrect guess, display the random number on the screen. Save and then run the program. Test the program appropriately.
6. Now modify the program so that it uses the `getRandomNumber` function shown in Figure 9-18 in the chapter. Save and then run the program. Test the program appropriately.
7. Finally, modify the program so that it uses the `getRandomNumber` function shown in Figure 9-19. Save and then run the program. Test the program appropriately.



LAB 9-2 Plan and Create

In this lab, you will plan and create an algorithm for Sydney Green. The problem specification along with sample calculations are shown in Figure 9-40.



You can use a calculator or a spreadsheet program (such as Microsoft Excel) to verify the payments shown in Figure 9-40 and also to perform your own calculations using the periodic payment formula.

Problem specification

While shopping for her dream car, Sydney Green has noticed that many auto dealers are offering buyers a choice of either a large cash rebate or an extremely low financing rate, much lower than the rate Sydney would pay by financing the car through her local credit union. Sydney is not sure whether to take the lower financing rate from the dealer or take the rebate and then finance the car through the credit union. She wants a program that will calculate and display her monthly car payment using both scenarios. The formula for calculating a periodic payment on a loan is shown below. In the formula, *principal* is the amount of the loan, *rate* is the periodic interest rate, and *term* is the number of periodic payments. Also shown below are two examples that use the formula to calculate a periodic payment. Example 1 calculates the annual payment for a $9000 loan for three years at 5% interest. The annual payment rounded to the nearest cent is $3304.88. In other words, if you borrow $9000 for three years at 5% interest, you would need to make three annual payments of $3304.88 to pay off the loan. Example 2 calculates the monthly payment for a $12,000 loan for five years at 6% interest. To pay off this loan, you would need to make 60 payments of $231.99. When calculating a monthly payment, you must convert the annual interest rate to a monthly interest rate; you do this by dividing the annual rate by 12. You also need to convert the term from years to months. This is accomplished by multiplying the number of years by 12. (When you apply for a loan, the lender typically quotes you an annual interest rate and expresses the term in years.)

Figure 9-40 Problem specification and sample calculations for Lab 9-2 *(continues)*

(continued)

Periodic payment formula
principal * *rate* / (1 – (*rate* + 1)$^{-term}$)

Example 1 calculates the annual payment for a loan of $9000 for 3 years at 5% interest

Principal:	9000
Annual rate:	.05
Term (years):	3
Formula:	9000 * .05 / (1 – (.05 + 1)$^{-3}$)
Annual payment:	$3304.88 (rounded to the nearest cent)

Example 2 calculates the monthly payment for a loan of $12,000 for 5 years at 6% interest

Principal:	12,000
Monthly rate:	.005 (annual rate of .06 divided by 12)
Term (months):	60 (5 years multiplied by 12)
Formula:	12,000 * .005 / (1 – (.005 + 1)$^{-60}$)
Monthly payment:	$231.99 (rounded to the nearest cent)

Figure 9-40 Problem specification and sample calculations for Lab 9-2

First, analyze the problem, looking for the output first and then for the input. In this case, Sydney wants the program to display two monthly payments: the payment if she finances the car through her credit union and the payment if she finances it through the dealer. To calculate the monthly payments, the computer will need to know the following information: the price of the car (after any trade-in), the rebate amount, the credit union's annual interest rate, the dealer's annual interest rate, and the term (in years). Next, plan the algorithm. Recall that most algorithms begin with an instruction to enter the input items into the computer, followed by instructions that process the input items, typically including the items in one or more calculations. Most algorithms end with one or more instructions that display, print, or store the output items. Figure 9-41 shows the completed IPO charts for the program's `main` and `getPayment` functions. Notice that the `main` function calls the value-returning `getPayment` function twice: once to calculate and return the credit union payment and again to calculate and return the dealer payment. The `getPayment` function uses the periodic payment formula to calculate the payments. To use the formula, the function needs to know three items of information: the principal, monthly rate, and number of months. These items will be passed to the `getPayment` function when it is invoked by a statement in the `main` function. When calling the `getPayment` function to calculate the credit union payment, the statement will pass the difference between the car price and the rebate as the principal. It also will pass the monthly credit union rate (which is the annual credit union rate divided by 12) and the number of months (which is the term times 12). Similarly, when calling the `getPayment` function to calculate the dealer payment, the `main` function will pass the car price as the principal, and also pass the monthly dealer rate and the number of months.

`main` function

Input	Processing	Output
car price rebate credit union rate (annual) dealer rate (annual) term (years)	Processing items: none	credit union payment dealer payment

Algorithm:
1. enter the car price, rebate, credit union rate, dealer rate, and term
2. call the getPayment function to calculate the credit union payment, pass the function the car price minus the rebate, the credit union rate / 12, and the term * 12
3. call the getPayment function to calculate the dealer payment, pass the function the car price, the dealer rate / 12, and the term * 12
4. display the credit union payment and the dealer payment

`getPayment` function

Input	Processing	Output
principal monthly rate number of months	Processing items: none	monthly payment

Algorithm:
1. calculate the monthly payment using the periodic payment formula
2. return the monthly payment

Figure 9-41 IPO charts for the `main` and `getPayment` functions

After completing the IPO chart, you then move on to the third step in the problem-solving process, which is to desk-check the algorithm. You will desk-check the algorithms in Figure 9-41 using $16000, $3000, .08, .03, and 4 as the car price (after any trade-in), rebate, credit union rate, dealer rate, and term (in years). Using these values, the monthly payments should be $317.37 (credit union) and $354.15 (dealer). Therefore, it will be cheaper for Sydney to finance the car through her credit union. Figure 9-42 shows the completed desk-check table.

`main` function

car price	rebate	credit union rate	dealer rate	term
16000	3000	.08	.03	4

credit union payment	dealer payment
317.37	354.15

`getPayment` function

principal	monthly rate	number of months	monthly payment
~~13000~~	~~.0067~~	~~48~~	~~317.37~~
16000	.0025	48	354.15

Figure 9-42 Completed desk-check table for the car payment algorithms

The fourth step in the problem-solving process is to code the algorithm into a program. The IPO chart information and C++ instructions for the `main` and `getPayment` functions are shown in Figures 9-43 and 9-44, respectively.



The variables declared in Figure 9-43 are local to the `main` function and remain in memory until the function ends.

`main` function

IPO chart information	C++ instructions
Input	
car price	`int carPrice = 0;`
rebate	`int rebate = 0;`
credit union rate (annual)	`double creditRate = 0.0;`
dealer rate (annual)	`double dealerRate = 0.0;`
term (years)	`int term = 0;`
Processing	
none	
Output	
credit union payment	`double creditPayment = 0.0;`
dealer payment	`double dealerPayment = 0.0;`
Algorithm	

1. enter the car price, rebate, credit union rate, dealer rate, and term	`cout << "Car price (after any trade-in): ";`
	`cin >> carPrice;`
	`cout << "Rebate: ";`
	`cin >> rebate;`
	`cout << "Credit union rate: ";`
	`cin >> creditRate;`
	`cout << "Dealer rate: ";`
	`cin >> dealerRate;`
	`cout << "Term in years: ";`
	`cin >> term;`

2. call the getPayment function to calculate the credit union payment, pass the function the car price minus the rebate, the credit union rate / 12, and the term * 12	`creditPayment =`
	`getPayment(carPrice -`
	`rebate, creditRate / 12,`
	`term * 12);`

3. call the getPayment function to calculate the dealer payment, pass the function the car price, the dealer rate / 12, and the term * 12	`dealerPayment =`
	`getPayment(carPrice,`
	`dealerRate / 12, term * 12);`

4. display the credit union payment and the dealer payment	`cout << "Credit union payment: $"`
	`<< creditPayment << endl;`
	`cout << "Dealer payment: $"`
	`<< dealerPayment << endl;`

Figure 9-43 IPO chart information and C++ code for the `main` function

The variables in Figure 9-44 are local to the `getPayment` function and remain in memory until the function ends.

`getPayment` function

IPO chart information	**C++ instructions**
Input	
principal (formal parameter)	`int prin`
monthly rate (formal parameter)	`double monthRate`
number of months (formal parameter)	`int months`
Processing	
none	
Output	
monthly payment	`double monthPay = 0.0;`
Algorithm	
1. calculate the monthly payment using the periodic payment formula	`monthPay = prin * monthRate / (1 - pow(monthRate + 1, -months));`
2. return the monthly payment	`return monthPay;`

Figure 9-44 IPO chart information and C++ code for the `getPayment` function

The fifth step in the problem-solving process is to desk-check the program. You begin by placing the names of the declared variables and named constants (if any) in a new desk-check table, along with their initial values. You then desk-check the remaining C++ instructions in order, recording in the desk-check table any changes made to the variables. Figure 9-45 shows the completed desk-check table for the car payment program. The results agree with those shown in the algorithm's desk-check table in Figure 9-42.

`main` function's variables

carPrice	rebate	creditRate	dealerRate	term
~~0~~	~~0~~	~~0.0~~	~~0.0~~	~~0~~
16000	3000	.08	.03	4

creditPayment	dealerPayment
~~0.0~~	~~0.0~~
317.37	354.15

`getPayment` function's variables

prin	monthRate	months	monthPay
~~13000~~	~~.0067~~	~~48~~	~~0.0~~
16000	.0025	48	~~317.37~~
			~~0.0~~
			354.15

Figure 9-45 Completed desk-check table for the car payment program

The final step in the problem-solving process is to evaluate and modify (if necessary) the program. Recall that you evaluate a program by entering its instructions into the computer and then using the computer to run (execute) it. While the program is running, you enter the same sample data used when desk-checking the program.

DIRECTIONS

Follow the instructions for starting your C++ development tool. Depending on the development tool you are using, you may need to create a new project; if so, name the project Lab9-2 Project and save it in the Cpp6\Chap09 folder. Enter the instructions shown in Figure 9-46 in a source file named Lab9-2.cpp. (Do not enter the line numbers.) Save the file in either the project folder or the Cpp6\Chap09 folder. Now, follow the appropriate instructions for running the Lab9-2.cpp file. Test the program using the same data you used to desk-check the program. Also test it using the data shown in Example 2 in Figure 9-40. If necessary, correct any bugs (errors) in the program.

```
 1  //Lab9-2.cpp - displays two monthly car payments
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <cmath>
 6  #include <iomanip>
 7  using namespace std;
 8
 9  //function prototype
10  double getPayment(int, double, int);
11
12  int main()
13  {
14     //declare variables
15     int carPrice         = 0;
16     int rebate           = 0;
17     double creditRate    = 0.0;
18     double dealerRate    = 0.0;
19     int term             = 0;
20     double creditPayment = 0.0;
21     double dealerPayment = 0.0;
22
23     //get input items
24     cout << "Car price (after any trade-in): ";
25     cin >> carPrice;
26     cout << "Rebate: ";
27     cin >> rebate;
28     cout << "Credit union rate: ";
29     cin >> creditRate;
30     cout << "Dealer rate: ";
31     cin >> dealerRate;
32     cout << "Term in years: ";
33     cin >> term;
34
35     //call function to calculate payments
36     creditPayment = getPayment(carPrice - rebate,
37          creditRate / 12, term * 12);
38     dealerPayment = getPayment(carPrice,
39          dealerRate / 12, term * 12);
40
```

the names of the formal parameters are not required

Figure 9-46 Car payment program *(continues)*

(continued)

```
41     //display payments
42     cout << fixed << setprecision(2) << endl;
43     cout << "Credit union payment: $"
44          << creditPayment << endl;
45     cout << "Dealer payment: $"
46          << dealerPayment << endl;
47
48     system("pause");
49     return 0;
50  }  //end of main function
51
52  //*****function definitions*****
53  double getPayment(int prin,
54                    double monthRate,
55                    int months)
56  {
57     //calculates and returns a monthly payment
58     double monthPay = 0.0;
59     monthPay = prin * monthRate /
60          (1 - pow(monthRate + 1, -months));
61     return monthPay;
62  }  //end of getPayment function
```

your C++ development tool may not require this statement

Figure 9-46 Car payment program



LAB 9-3 Modify

If necessary, create a new project named Lab9-3 Project. Enter (or copy) the Lab9-2.cpp instructions into a new source file named Lab9-3.cpp. Change Lab9-2.cpp in the first comment to Lab9-3.cpp. Make the following three modifications to the program. First, allow the user to enter the interest rates either as a whole number or as a decimal number. For example, if the interest rate is 5%, the user should be able to enter either 5 or .05. Second, the program should compare both monthly payments and then display one of the following three messages: "Take the rebate and finance through the credit union.", "Don't take the rebate. Finance through the dealer.", or "You can finance through either one." Third, the user should be able to calculate the monthly payments as many times as needed without having to run the program again. Save and then run the program. Test the program appropriately.



LAB 9-4 Desk-Check

Desk-check the code in Figure 9-47 using the data shown below. What current balance will the code display on the screen?

Beginning balance: 2000

w, 400, y

D, 1200, y

W, 45, y

w, 55, y

k, y

w, 150, y

d, 15, y

W, 1050, n

```
 1  //Lab9-4.cpp - displays an ending balance
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <iomanip>
 6  using namespace std;
 7
 8  //function prototype
 9  double getBalance(double, char, double);
10
11  int main()
12  {
13     //declare variables
14     double balance   = 0.0;
15     double amount    = 0.0;
16     char transaction = ' ';
17     char another     = 'Y';
18
19     cout << "Beginning balance: ";
20     cin >> balance;
21
22     do
23     {
24         //get input items
25         cout << "Withdrawal or Deposit? Enter W or D: ";
26         cin >> transaction;
27         transaction = toupper(transaction);
28
```

Figure 9-47 Code for Lab 9-4 *(continues)*

(continued)

```
29          if (transaction == 'W' || transaction == 'D')
30          {
31              cout << "Enter amount: ";
32              cin >> amount;
33              //call function to calculate balance
34              balance =
35                  getBalance(balance, transaction, amount);
36          }
37          else
38              cout << "Incorrect transaction type";
39          //end if
40
41          cout << endl << "Another transaction (Y/N)? ";
42          cin >> another;
43          cout << endl;
44      } while (toupper(another) == 'Y');
45
46      //display balance
47      cout << fixed << setprecision(2) << endl;
48      cout << "Current balance: $" << balance << endl;
49
50      system("pause");
51      return 0;
52  }  //end of main function
53
54  //*****function definitions*****
55  double getBalance(double bal, char type, double amt)
56  {
57      //calculates and returns the current balance
58      double curBalance = 0.0;
59      if (type == 'W')
60          curBalance = bal - amt;
61      else
62          curBalance = bal + amt;
63      //end if
64      return curBalance;
65  }  //end of getBalance function
```

your C++ development tool may not require this statement

Figure 9-47 Code for Lab 9-4



LAB 9-5 Debug

Follow the instructions for starting C++ and opening the Lab9-5.cpp file. The file is contained in either the Cpp6\Chap09\Lab9-5 Project folder or the Cpp6\Chap09 folder. Test the program using 20500, 3500, and 10 as the asset cost, salvage value, and useful life. The depreciation should be $1700.00. Debug the program.

Summary

- Functions allow a programmer to avoid duplicating code in different parts of a program. They also allow large and complex programs to be broken into small and manageable tasks.
- Some of the functions used in a program are built-in functions. Others, like `main`, are program-defined functions.
- All functions are classified as either value-returning functions or void functions. A value-returning function returns precisely one value after completing its assigned task. The value is returned to the statement that called the function. Void functions, which you will learn about in Chapter 10, do not return a value.
- You can use the C++ built-in value-returning `sqrt` function to find the square root of a number. The function returns the square root as a `double` number. A program that uses the `sqrt` function must contain the `#include <cmath>` directive.
- The items within parentheses in a function call are referred to as actual arguments.
- The C++ language provides the `rand` function for generating random numbers. The `rand` function is a value-returning function. It returns an integer that is greater than or equal to zero but less than or equal to `RAND_MAX`, whose value is always at least 32767. You can use the expression *lowerBound* **+ rand() % (***upperBound* **–** *lowerBound* **+ 1)** to produce random integers within a specific range.
- You can initialize the `rand` function using the C++ built-in void `srand` function. Most programmers use the built-in value-returning `time` function as the `srand` function's *seed* argument. A program that uses the `time` function must contain the `#include <ctime>` directive.
- A function definition is composed of a function header and a function body.
- The function header is the first line in the function definition. The function header specifies the type of data the function returns, as well as the name of the function and an optional *parameterList* enclosed in parentheses. The items listed in the *parameterList* are called formal parameters.
- The *parameterList* in a function header contains the data type and name of each formal parameter. The quantity, data type, and sequence of the formal parameters in the *parameterList* should agree with the quantity, data type, and sequence of the actual arguments passed to the function. In most cases, the name of each formal parameter is different from the name of its corresponding actual argument. Functions that do not require a *parameterList* will have an empty set of parentheses after the function's name.
- The function body in a function definition contains the instructions that the function must follow to perform its assigned task. The function body begins with an opening brace and ends with a closing brace. Typically, the `return` statement, which instructs the function to return a value, is the last statement in the function body of a value-returning function.

- You call a function by including its name and actual arguments (if any) in a statement.
- Unless specified otherwise, variables in C++ are passed to a function *by value*, which means that only the value stored in the variable is passed.
- A program will have one function prototype for each function defined below the `main` function. Functions defined above the `main` function in a program do not need a function prototype.
- A variable's scope, which can be either local or global, indicates where in a program a variable can be used. A variable's lifetime indicates how long the variable remains in the computer's internal memory.
- Local variables can be used only within the function in which they are declared or in whose *parameterList* they appear, and they remain in memory until the function ends. Global variables, which you should avoid using, can be used anywhere in the program. Unlike local variables, global variables remain in memory until the program ends.
- If more than one memory location has the same name and the name appears in a statement, the computer uses the position of the statement within the program to determine which memory location to use. For clarity, you should use unique variable names within a program.

Key Terms

Actual argument—an item of information passed (sent) to a function when the function is called (invoked)

Built-in functions—blocks of code that perform a task and are included in libraries that come with the C++ language; examples include the `pow`, `sqrt`, `rand`, `srand`, and `time` functions

Formal parameters—the memory locations listed in a function header's *parameterList*; a formal parameter stores an item of information passed to a function when the function is invoked (called)

Function prototype—a statement that specifies the function's name, the data type of its return value (if any), and the data type and (optionally) name of each of its formal parameters (if any); required for every function that is defined below the `main` function in a program

Global variables—variables that are declared outside of any function in a program; global variables can be used by any statement below the variable declaration in the program, and they remain in memory until the program ends; you should avoid using global variables in a program

Lifetime—indicates how long an item, such as a variable, remains in the computer's internal memory

Local variables—variables that are either declared within a function or appear in the function header's *parameterList*; local variables can be used only by the function in which they are declared or in whose *parameterList* they appear; local variables remain in memory until the function ends

Passing *by reference*—refers to the process of passing a variable's address to a function

Passing *by value*—refers to the process of passing a variable's value to a function

Program-defined functions—blocks of code that perform a task and are written by a programmer; usually the task will avoid the duplication of code within a program or perform some common task

Pseudo-random number generator—a device that produces a sequence of numbers that meet certain statistical requirements for randomness; the `rand` function is the pseudo-random number generator in C++

`rand` function—a built-in C++ function that returns a random integer that is greater than or equal to zero but less than or equal to the value stored in the `RAND_MAX` constant; the pseudo-random number generator in C++

`RAND_MAX`—a C++ built-in constant that represents the largest integer generated by the `rand` function; although the value of `RAND_MAX` varies with different computer systems, its value is always at least 32767

`return` statement—in most cases, the last statement in a value-returning function; it alerts the computer that the function has completed its task

Scope—indicates where in the program an item, such as a variable, can be used

`sqrt` function—a C++ built-in function whose purpose is to return the square root of a number that has either the `double` or `float` data type; returns the square root as a `double` number; a program that uses the `sqrt` function must contain the `#include <cmath>` directive

`srand` function—a C++ built-in function used to initialize the `rand` function

`time` function—a built-in C++ function that returns the current time (according to your computer system's clock) as seconds elapsed since midnight on January 1, 1970; often used as the *seed* argument in the `srand` function; a program that uses the `time` function must contain the `#include <ctime>` directive

Value-returning functions—functions that return precisely one value after they complete their assigned task

Review Questions

1. Value-returning functions can return _____.
   a. one value only
   b. one or more values
   c. the number 0 only
   d. none of the above

2. The function header specifies _____.
   a. the data type of the function's return value (if any)
   b. the name of the function
   c. the function's formal parameters (if any)
   d. all of the above

3. Which of the following is false?
   a. The number of actual arguments should agree with the number of formal parameters.
   b. The data type of each actual argument should match the data type of its corresponding formal parameter.
   c. The name of each actual argument should be identical to the name of its corresponding formal parameter.
   d. When you pass information to a function *by value*, the function stores the value of each item it receives in a separate memory location.

4. Each memory location listed in a function header's *parameterList* is referred to as _____.
   a. an actual argument
   b. an actual parameter
   c. a formal argument
   d. a formal parameter

5. A program contains the statement `tax = calcTax(sales);`. The `tax` and `sales` variables have the `double` data type. Which of the following is a valid function header for the `calcTax` function?
   a. `calcTax(double sales);`
   b. `double calcTax(salesAmount)`
   c. `double calcTax(double salesAmount)`
   d. `double calcTax(int sales);`

6. Which of the following is a valid function header for the `getFee` function, which receives an integer first and a number with a decimal place second? The function returns a number with a decimal place.
   a. `getFee(int base, double rate);`
   b. `double getFee(int base, double rate);`
   c. `double getFee(double base, int rate)`
   d. `double getFee(int base, double rate)`

7. Which of the following is a valid function prototype for the function described in Review Question 6?
   a. `getFee(int base, double rate);`
   b. `int getFee(int, double)`
   c. `double getFee(int base, double rate)`
   d. `double getFee(int, double);`

8. Which of the following directs a function to return the contents of the `stateTax` variable to a statement contained in the `main` function?
   a. `restore stateTax;`
   b. `return stateTax`
   c. `return to main(stateTax);`
   d. none of the above

9. If the statement `netPay = calcNet(gross, taxes);` passes the contents of the `gross` and `taxes` variables to the `calcNet` function, the variables are said to be passed _____.
   a. *by address*
   b. *by content*
   c. *by reference*
   d. *by value*

10. A variable's _____ indicates where in the program a variable can be used.
    a. lifetime
    b. range
    c. scope
    d. span

11. If a variable named `beginBalance` appears in a function header's *parameterList*, which of the following statements is true?
    a. The `beginBalance` variable remains in memory until the function ends.
    b. The `beginBalance` variable is called a functional variable.
    c. The `beginBalance` variable can be used anywhere in the program.
    d. both a and b

12. A program contains three functions named `main`, `calcGross`, and `displayGross`. Two of the functions—`main` and `calcGross`—declare a variable named `pay`. The `pay` variable name also appears in the `displayGross` function header. When the computer processes the statement `pay = hours * rate;` in the `calcGross` function, it multiplies the contents of the `hours` variable by the contents of the `rate` variable. It then stores the result in which function's `pay` variable?
    a. `calcGross`
    b. `displayGross`
    c. `main`
    d. none of the above, because you can't have more than one memory location with the same name

13. Which of the following expressions produces a random integer from 3 to 9, inclusive?

    a. `1 + rand() % (9 - 3 + 1)`

    b. `3 + rand() % (9 - 3 + 1)`

    c. `3 + rand() % (9 + 3 - 1)`

    d. `9 + rand() % (9 + 1 - 3)`

14. Which of the following can be used to initialize the random number generator in C++?

    a. `init(static_cast<int>(time(0)));`

    b. `rand(static_cast<int>(time(0)));`

    c. `srand(static_cast<int>(time(0)));`

    d. none of the above

15. A program that uses the `sqrt` function must contain the _____ directive.

    a. `#include <cmath>`

    b. `#include <ctime>`

    c. `#include <square>`

    d. `#include <squareRoot>`

Exercises



Pencil and Paper

TRY THIS

1. Write the C++ code for a function that receives an integer passed to it. The function should divide the integer by 2 and then return the result, which may contain a decimal place. Name the function `divideByTwo`. Name the formal parameter `wholeNumber`. (The answers to TRY THIS Exercises are located at the end of the chapter.)

TRY THIS

2. Write the function prototype for the `divideByTwo` function from Pencil and Paper Exercise 1. (The answers to TRY THIS Exercises are located at the end of the chapter.)

TRY THIS

3. Write a statement that calls the `divideByTwo` function from Pencil and Paper Exercise 1, passing the function the contents of the `total` variable. The statement should assign the function's return value to a `double` variable named `quotient`. (The answers to TRY THIS Exercises are located at the end of the chapter.)

4. Rewrite the code from Pencil and Paper Exercises 1, 2, and 3 so that the `divideByTwo` function receives two integers rather than one integer. The function should add together both integers and then divide the sum by 2. Here again, the function's return value may contain a decimal place. Name the formal parameters `num1` and `num2`. Name the actual arguments `total1` and `total2`. MODIFY THIS

5. Write a C++ statement that displays a random integer from 50 through 100 on the computer screen. INTRODUCTORY

6. Write a C++ statement that assigns the square root of a number to a `double` variable named `sqRoot`. The number is stored in a `double` variable named `num`. INTRODUCTORY

7. Write the C++ code for a function that prompts the user to enter a character and then stores the character in a `char` variable named `response`. The function should return the contents of the `response` variable. Name the function `getCharacter`. (The function will not have any actual arguments passed to it.) Also write an appropriate function prototype for the `getCharacter` function. In addition, write a statement that invokes the `getCharacter` function and assigns its return value to a `char` variable named `custCode`. INTRODUCTORY

8. Write the C++ code for a function that receives four `double` numbers. The function should calculate the average of the four numbers and then return the result. Name the function `calcAverage`. Name the formal parameters `num1`, `num2`, `num3`, and `num4`. Also write an appropriate function prototype for the `calcAverage` function. In addition, write a statement that invokes the `calcAverage` function and assigns its return value to a `double` variable named `quotient`. Use the following numbers as the actual arguments: 45.67, 8.35, 125.78, and 99.56. INTERMEDIATE

9. Write a C++ statement that adds the cube of the number stored in the `num1` variable to the square root of the number stored in the `num2` variable. The statement should assign the result to the `answer` variable. All of the variables have the `double` data type. INTERMEDIATE

10. Write a C++ statement that assigns to the `answer` variable the square root of the following expression: $x^2 * y^3$. The `x`, `y`, and `answer` variables have the `double` data type. INTERMEDIATE

11. Write a C++ statement that assigns to the `rate` variable the result of the following expression: $(future / present)^{1 - term} - 1$. The three variables have the `double` data type. ADVANCED

12. A program's `main` function declares three `double` variables named `salesTax`, `sales`, and `taxRate`. It also declares a `char` variable named `status`. The `main` function contains the following statement: `salesTax = getSalesTax(sales, status, taxRate);`. The statement calls the `getSalesTax` function, whose function header is `int getSalesTax(char code, int sold, double rate)`. Correct the function header. SWAT THE BUGS

Computer

TRY THIS

13. If necessary, create a new project named TryThis13 Project. Enter the C++ instructions from Figure 9-4 into a source file named TryThis13.cpp. Change the filename in the first comment. Save and then run the program. Test the program using the data shown in Figure 9-5 in the chapter. (The answers to TRY THIS Exercises are located at the end of the chapter.)

TRY THIS

14. If necessary, create a new project named TryThis14 Project. Code the IPO charts shown in Figure 9-48. If necessary, create a new project named TryThis14 Project. Enter your C++ instructions into a source file named TryThis14.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Display the Celsius temperature in fixed-point notation with no decimal places. Save and then run the program. Test the program using the following Fahrenheit temperatures: 32 and 212. (The answers to TRY THIS Exercises are located at the end of the chapter.)

`main` function

Input	Processing	Output
Fahrenheit temperature	Processing items: none	Celsius temperature
	Algorithm: 1. call getFahrenheit to get the Fahrenheit temperature 2. call calcCelsius to calculate the Celsius temperature, pass the Fahrenheit temperature 3. display the Celsius temperature	

`getFahrenheit` function

Input	Processing	Output
Fahrenheit temperature	Processing items: none	Fahrenheit temperature
	Algorithm: 1. enter the Fahrenheit temperature 2. return the Fahrenheit temperature	

`calcCelsius` function

Input	Processing	Output
Fahrenheit temperature	Processing items: none	Celsius temperature
	Algorithm: 1. Celsius temperature = 5.0 / 9.0 * (Fahrenheit temperature - 32.0) 2. return the Celsius temperature	

Figure 9-48

MODIFY THIS

15. In this exercise, you modify the code from Computer Exercise 13. If necessary, create a new project named ModifyThis15 Project. Enter (or copy) the TryThis13.cpp instructions into a new source file

named ModifyThis15.cpp. Change TryThis13.cpp in the first comment to ModifyThis15.cpp. Remove both calculation tasks from the `main` function and assign both to a program-defined value-returning function named `getHypotenuse`. Save and then run the program. Test the program appropriately.

16. In this exercise, you modify the code from Computer Exercise 14. If necessary, create a new project named ModifyThis16 Project. Enter (or copy) the TryThis14.cpp instructions into a new source file named ModifyThis16.cpp. Change TryThis14.cpp in the first comment to ModifyThis16.cpp. Modify the program so that the user can convert as many temperatures as desired without having to run the program again. Save and then run the program. Test the program appropriately. MODIFY THIS

17. If necessary, create a new project named Introductory17 Project. Enter the C++ instructions shown in Figure 9-19 into a source file named Introductory17.cpp. Change the filename in the first comment. Save and then run the program. Modify the program so that it performs subtraction rather than addition. However, the program should always subtract the smaller number from the larger one. Save and then run the program. INTRODUCTORY

18. In this exercise, you modify the program from Lab 7-2 in Chapter 7. If necessary, create a new project named Introductory18 Project. Copy the instructions from the Lab7-2.cpp file into a source file named Introductory18.cpp. (Alternatively, you can enter the instructions from Figure 7-48 into the Introductory18.cpp file.) Change the filename in the first comment. Modify the program so that it uses a value-returning function to determine the grade. Save and then run the program. Test the program appropriately. INTRODUCTORY

19. In this exercise, you experiment with the concepts of scope and lifetime. INTERMEDIATE

    a. Follow the instructions for starting C++ and opening the Intermediate19.cpp file. The file is contained in either the Cpp6\Chap09\Intermediate19 Project folder or the Cpp6\Chap09 folder. Run the program. If you are asked whether you want to run the last successful build, click the No button. The C++ compiler displays an error message indicating that the `getDoubleNumber` function does not recognize the `number` variable. The error occurs because the `number` variable is local to the `main` function. To fix this error, you can either pass the `number` variable's value to the `getDoubleNumber` function or create a global variable named `number`. Passing the variable's value is the preferred way for the `main` function to communicate with the `getDoubleNumber` function. However, to give you an opportunity to see how global variables work in a program, you will fix the program's error by creating a global variable named `number`.

    b. Change the `int number = 0;` statement in the `main` function to a comment. Recall that global variables are declared outside of any function in the program. In the blank line below the `//declare global variable` comment, type `int number = 0;`. Because the `number` variable is now a global variable, both the `main` and `getDoubleNumber` functions have access to it. You will run the program to verify that fact.

c. Save and then run the program. When prompted for a number, type 5 and press Enter. The Command Prompt window shows that doubling the number 5 results in the number 10, which is correct.

d. Now change the statement that declares the global `number` variable to a comment. Also remove the two forward slashes from the `//int number = 0;` line in the `main` function. Fix the program's error by passing the value contained in the `number` variable to the `getDoubleNumber` function.

e. Save and then run the program. When prompted for a number, type 5 and press Enter. The Command Prompt window shows that doubling the number 5 results in the number 10, which is correct.

INTERMEDIATE

20. In this exercise, you modify the program from Lab 6-2 in Chapter 6. If necessary, create a new project named Intermediate20 Project. Copy the instructions from the Lab6-2.cpp file into a source file named Intermediate20.cpp. (Alternatively, you can enter the instructions from Figure 6-32 into the Intermediate20.cpp file.) Change the filename in the first comment. Modify the program so that it uses a value-returning function to determine the commission. Save and then run the program. Test the program appropriately.

INTERMEDIATE

21. In this exercise, you modify the program from Lab 5-2 in Chapter 5. If necessary, create a new project named Intermediate21 Project. Copy the instructions from the Lab5-2.cpp file into a source file named Intermediate21.cpp. (Alternatively, you can enter the instructions from Figure 5-33 into the Intermediate21.cpp file.) Change the filename in the first comment. Modify the program so that it uses two value-returning functions: one to determine the fat calories and the other to determine the fat percentage. Save and then run the program. Test the program appropriately.

INTERMEDIATE

22. The payroll manager at Gerston Blankets wants a program that calculates and displays the gross pay for each of the company's employees. It also should calculate and display the total gross pay. The payroll manager will enter the number of hours the employee worked and his or her pay rate. Employees working more than 40 hours should receive time and one-half for the hours over 40. Use a value-returning function to determine an employee's gross pay. Use a different value-returning function to accumulate the total gross pay. The program should display the total gross pay only after the payroll manager has finished entering the data for all the employees. Use a sentinel value to end the program.

a. Create IPO charts for the problem, and then desk-check the algorithm using the following four sets of hours worked and pay rates: 35, $10.50; 43, $15; 32, $9.75; 20, $6.45.

b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 9-48. Then code the algorithm into a program.

c. Desk-check the program using the same data used to desk-check the algorithm.

d. If necessary, create a new project named Intermediate22 Project. Enter your C++ instructions into a source file named Intermediate22.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

e. Save and then run the program. Test the program using the same data used to desk-check the program.

23. ADVANCED

In this exercise, you create a program that calculates the average of three test scores. The program should contain three value-returning functions: `main`, `getTestScore`, and `calcAverage`. The `main` function should call the `getTestScore` function to get and return each of three test scores. The test scores may contain a decimal place. (Hint: The `main` function will need to call the `getTestScore` function three times.) The `main` function then should call the `calcAverage` function to calculate and return the average of the three test scores. When the `calcAverage` function has completed its task, the `main` function should display the average on the screen. Display the average with one decimal place.

a. Create IPO charts for the problem, and then desk-check the algorithm using the following four sets of test scores: 56, 78, 90; 100, 85, 67; 74, 32, 98; 25, 99, 84.

b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 9-48. Then code the algorithm into a program.

c. Desk-check the program using the same data used to desk-check the algorithm.

d. If necessary, create a new project named Advanced23 Project. Enter your C++ instructions into a source file named Advanced23.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

e. Save and then run the program. Test the program using the same data used to desk-check the program.

24. ADVANCED

In this exercise, you create a program that calculates and displays gross pay amounts. The user will enter the number of hours an employee worked and his or her pay rate. The program should contain four value-returning functions: `main`, `getHoursWorked`, `getPayRate`, and `calcGross`. The `main` function should call each of the other three functions and then display the gross pay on the screen. When coding the `calcGross` function, you do not have to worry about overtime pay. You can assume that everyone works 40 or fewer hours per week. The hours worked and rate of pay may contain a decimal place. Use a sentinel value to end the program.

a. Create IPO charts for the problem, and then desk-check the algorithm using the following two sets of hours worked and pay rates: 25.5, $12; 40, $11.55.

b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 9-48. Then code the algorithm into a program.

c. Desk-check the program using the same data used to desk-check the algorithm.

d. If necessary, create a new project named Advanced24 Project. Enter your C++ instructions into a source file named Advanced24.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

e. Save and then run the program. Test the program using the same data used to desk-check the program.

ADVANCED

25. In this exercise, you learn about using a default value for a formal parameter.

a. You can define a default value for one or more of a function's formal parameters. If a formal parameter has a default value, then you do not need to provide an actual argument for it when you call the function. Follow the instructions for starting C++ and opening the Advanced25.cpp file. The file is contained in either the Cpp6\Chap09\Advanced25 Project folder or the Cpp6\Chap09 folder. The program uses a value-returning function to calculate a bonus amount. The bonus rate is based on the salesperson's code: either A or B. The user enters the bonus rate only for salespeople with an "A" code. Salespeople with a "B" code always receive a 5% bonus.

b. You define a default value for a formal parameter in the function's prototype, using the syntax *formalParameterDataType = defaultValue*. If the prototype also contains the parameter's name, you use the syntax *formalParameterDataType formalParameterName = defaultValue*. Keep in mind that all formal parameters having a default value must be placed after those that do not have a default value in the function prototype. Change the function prototype so that it uses the number .05 as the default value for the bonus rate.

c. Save and then run the program. Test the program appropriately.

SWAT THE BUGS

26. Follow the instructions for starting C++ and opening the SwatTheBugs26.cpp file. The file is contained in either the Cpp6\Chap09\SwatTheBugs26 Project folder or the Cpp6\Chap09 folder. Debug the program.

Answers to TRY THIS Exercises



Pencil and Paper

1.
```
double divideByTwo(int wholeNumber)
{
     return wholeNumber / 2;
}    //end of divideByTwo function
```

2. `double divideByTwo(int);` or `double divideByTwo(int wholeNumber);`

3. `quotient = divideByTwo(total);`



Computer

13. No answer required.

16. See Figure 9-49.

```
 1  //TryThis14.cpp - converts Fahrenheit to Celsius
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <iomanip>
 6  using namespace std;
 7
 8  //function prototypes
 9  int getFahrenheit();
10  double calcCelsius(int tempF);
11
12  int main()
13  {
14     int fahrenheit = 0;
15     double celsius = 0.0;
16
17     //get input item
18     fahrenheit = getFahrenheit();
19
20     //calculate Celsius
21     celsius = calcCelsius(fahrenheit);
22
23     //display output item
24     cout << fixed << setprecision(0);
25     cout << "Celsius: " << celsius << endl;
26
27     system("pause");
28     return 0;
29  }  //end of main function
30
31  //*****function definitions*****
32  int getFahrenheit()
33  {
34     int tempF = 0;
35     cout << "Enter Fahrenheit temperature: ";
36     cin >> tempF;
37     return tempF;
38  }  //end of getFahrenheit function
39
40  double calcCelsius(int tempF)
41  {
42     double tempC = 0.0;
43     tempC = 5.0 / 9.0 * (tempF - 32.0);
44     return tempC;
45  }  //end of calcCelsius function
```

your C++ development tool may not require this statement

Figure 9-49

CHAPTER 10

Void Functions

After studying Chapter 10, you should be able to:

- Create a function that does not return a value
- Invoke a function that does not return a value
- Pass information *by reference* to a function

Void Functions

As discussed in Chapter 9, all built-in and program-defined functions are categorized as either value-returning functions or void functions. Recall that a value-returning function performs a task and then returns precisely one value to the statement that called it. You learned how to create and invoke value-returning functions in Chapter 9. Like value-returning functions, void functions also perform a task. However, unlike value-returning functions, **void functions** do not return a value after completing their task. You already are familiar with one void function: `srand`. Recall from Chapter 9 that the `srand` function is a built-in void function whose task is to initialize the C++ random number generator. In this chapter, you will learn how to create program-defined void functions. A program might use a program-defined void function to display information (such as a title and column headings) at the top of each page in a report. Rather than duplicating the required code several times in the program, the code can be entered once in a void function. The program then can call the void function whenever and wherever it is needed. A void function is appropriate in this situation because the function does not need to return a value after completing its task. Figure 10-1 shows the syntax used to create a void function in a C++ program. When you compare this syntax with the one for creating a value-returning function (shown in Figure 9-14 in Chapter 9), you will notice two differences. First, a void function's header begins with the keyword `void` rather than with a data type. The `void` keyword indicates that the function does not return a value. Second, the function body in a void function does not contain a `return` statement, which is required in the function body of a value-returning function. The `return` statement is not necessary in a void function body because a void function does not return a value. Also included in Figure 10-1 are examples of program-defined void functions.

HOW TO Create a Program-Defined Void Function

Syntax

```
void functionName([parameterList])
{
        one or more statements
}       //end of functionName function
```

function header

function body

Example 1

```
void displayLine()
{
        cout << "--------------------" << endl;
}       //end of displayLine function
```

function definition

The function displays a straight line composed of 20 hyphens.

Figure 10-1 How to create a program-defined void function *(continues)*

(continued)

Example 2

```
void displayCompanyInfo()
{
      cout << "ABC Company" << endl;
      cout << "Chicago, Illinois" << endl;
}     //end of displayCompanyInfo function
```

The function displays a company's name, city, and state.

Example 3

```
void displayTotalSales(int total)
{
      cout << "Total sales: $" << total << endl;
}     //end of displayTotalSales function
```

The function displays the total sales it receives from the statement that invoked it.

Figure 10-1 How to create a program-defined void function

Figure 10-2 shows the IPO chart information and C++ instructions for the ABC Company program, which uses the void functions from Figure 10-1. Figure 10-3 shows the program's code, with the function prototypes and calls shaded. As you do with a value-returning function, you call a void function by including its name and actual arguments (if any) in a statement. However, unlike a call to a value-returning function, a call to a void function appears as a statement by itself rather than as part of another statement. Each call to the void functions in Figure 10-3, for example, is a self-contained statement. When the computer processes a statement that calls a program-defined void function, the computer first locates the function's code in the program. If the function call contains an *argumentList*, the computer passes the values of the actual arguments (assuming the variables included in the *argumentList* are passed *by value*) to the called function. The function receives the values and stores them in the formal parameters listed in its *parameterList*. Then, the computer processes the function's code. When the function ends, the computer continues program execution with the statement immediately below the one that called the function. In the program shown in Figure 10-3, for example, the `displayLine();` statement on Line 28 calls the `displayLine` function. After processing the `displayLine` function's code, the computer returns to the `main` function to process the statement on Line 29; that statement calls the `displayCompanyInfo` function. When the computer finishes processing the code in the `displayCompanyInfo` function, it returns to the `main` function to process the statement on Line 30; that statement calls the `displayTotalSales` function. After processing the code in the `displayTotalSales` function, the computer returns to the `main` function to process the `displayLine();` statement on Line 31. After the `displayLine` function completes its task, the computer returns to the `main` function to process the `system("pause");` statement on Line 33. Figure 10-4 shows a sample run of the program. The sample run contains output from each of the four functions in the program.



As you learned in Chapter 9, value-returning functions typically are called from statements that do one of the following: display the function's return value, use the return value in a calculation or comparison, or assign the return value to a variable.

main function

IPO chart information	C++ instructions
Input	
store 1's sales	`int store1Sales = 0;`
store 2's sales	`int store2Sales = 0;`
Processing	
none	
Output	
total sales	`int totalSales = 0;` (displayed by the displayTotalSales function)
straight line (2 of them)	displayed by the displayLine function
name , city, and state	displayed by the displayCompanyInfo function
Algorithm	

1. enter store 1's sales and store 2's sales	`cout << "Store 1's sales: ";`
	`cin >> store1Sales;`
	`cout << "Store 2's sales: ";`
	`cin >> store2Sales;`

2. calculate the total sales by adding together store 1's sales and store 2's sales	`totalSales = store1Sales + store2Sales;`
3. call the displayLine function to display a straight line	`displayLine();`
4. call the displayCompanyInfo function to display the name, city, and state	`displayCompanyInfo();`
5. call the displayTotalSales function to display the total sales, pass the total sales to the function	`displayTotalSales(totalSales);`
6. call the displayLine function to display a straight line	`displayLine();`

displayLine function

IPO chart information	C++ instructions
Input	
none	
Processing	
none	
Output	
straight line (composed of 20 hyphens)	displayed using a string literal constant

Figure 10-2 IPO chart information and C++ instructions for the ABC Company program *(continues)*

(continued)

Algorithm
display a straight line

```
cout << "--------------------"
<< endl;
```

displayCompanyInfo function

IPO chart information	**C++ instructions**
Input	
none	
Processing	
none	
Output	
name, city, and state	*displayed using string literal constants*

Algorithm
display name, city, and state

```
cout << "ABC Company" << endl;
cout << "Chicago, Illinois"
<< endl << endl;
```

displayTotalSales function

IPO chart information	**C++ instructions**
Input	
total sales (formal parameter)	`int total`
Processing	
none	
Output	
total sales	

Algorithm
display total sales

```
cout << "Total sales: $"
<< total << endl;
```

Figure 10-2 IPO chart information and C++ instructions for the ABC Company program

```
 1  //ABC.cpp - displays the total sales
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  using namespace std;
 6
 7  //function prototypes
 8  void displayLine();
 9  void displayCompanyInfo();
10  void displayTotalSales(int total);
11
12  int main()
13  {
14      int store1Sales = 0;
15      int store2Sales = 0;
```

function prototypes end with a semicolon

Figure 10-3 ABC Company program *(continues)*

(continued)

```
16      int totalSales  = 0;
17
18      //enter input items
19      cout << "Store 1's sales: ";
20      cin >> store1Sales;
21      cout << "Store 2's sales: ";
22      cin >> store2Sales;
23
24      //calculate total sales
25      totalSales = store1Sales + store2Sales;
26
27      //display output items
28      displayLine();
29      displayCompanyInfo();
30      displayTotalSales(totalSales);
31      displayLine();
32
33      system("pause");
34      return 0;
35  }   //end of main function
36
37  //*****function definitions*****
38  void displayLine()
39  {
40      cout << "--------------------" << endl;
41  }   //end of displayLine function
42
43  void displayCompanyInfo()
44  {
45      cout << "ABC Company" << endl;
46      cout << "Chicago, Illinois" << endl << endl;
47  }   //end of displayCompanyInfo function
48
49  void displayTotalSales(int total)
50  {
51      cout << "Total sales: $" << total << endl;
52  }   //end of displayTotalSales function
```

your C++ development tool may not require this statement

function headers do not end with a semicolon

Figure 10-3 ABC Company program

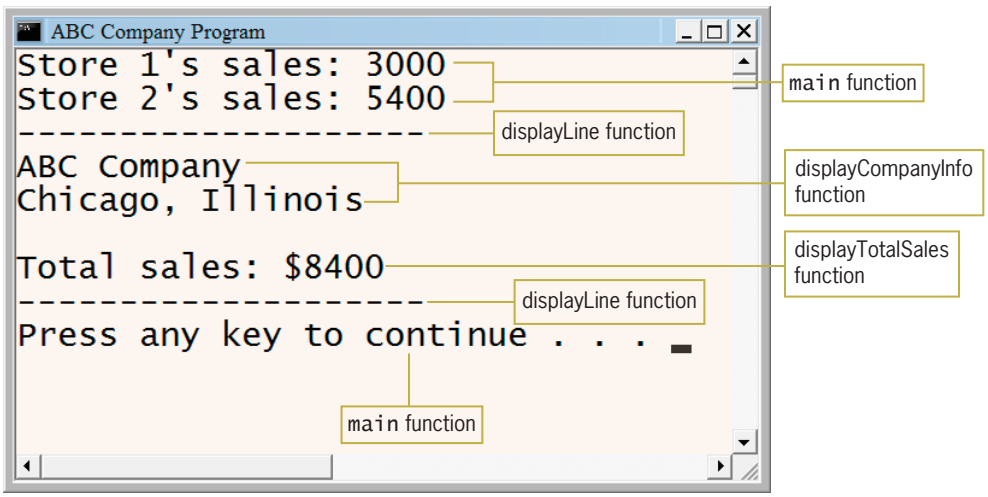


Figure 10-4 Sample run of the ABC Company program



The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 10-1

1. In C++, the function header for a function that does not return a value begins with the keyword _____.

2. Write a C++ statement that calls a void function named `displayTaxes`. The statement passes the contents of two `double` variables named `federalTax` and `localTax`.

3. Write the function header for the `displayTaxes` function from Question 2. Use `fedTax` and `stateTax` as the names for the formal parameters.

4. The `return` statement typically is the last statement in a C++ void function.

   a. True

   b. False

Passing Variables to a Function



The internal memory of a computer is similar to a large post office. Like each post office box, each memory cell has a unique address.



Only variables can be passed *by reference*.

As you learned in Chapter 9, the items passed to a function are called actual arguments. An actual argument can be a variable, named constant, literal constant, or keyword; however, in most cases, it will be a variable. Recall that each variable declared in a program has both a value and a unique address that represents the location of the variable in the computer's internal memory. C++ allows you to pass either the variable's value or its address to a function. Passing a variable's value is referred to as passing *by value*, whereas passing its address is referred to as **passing *by reference***. The method you choose—*by value* or *by reference*—depends on whether you want the receiving function to have access to the variable in memory. In other words, it depends on whether you want to allow the receiving function to change the contents of the variable. Although the idea of passing information *by value* and *by reference* may sound confusing at first, it is a concept with which you already are familiar. To illustrate, assume you have a savings account at a local bank. During a conversation with your friend Melissa, you mention the amount of money you have in the account. Sharing this information with Melissa is similar to passing a variable *by value*. Knowing the balance in your account does not give Melissa access to your bank account. It merely provides information that she can use to compare with the amount of money she has saved. The savings account example also provides an illustration of passing information *by reference*. To either deposit money in your account or withdraw money from your account, you must provide the bank teller with your account number. The account number represents the location of your account at the bank and allows the teller to change the account balance. Giving the teller your bank account number is similar to passing a variable *by reference*. The account number allows the teller to change the contents of your bank account, similar to the way a variable's address allows the receiving function to change the contents of the variable. Before learning how to pass a variable *by reference*, you will review the concept of passing *by value*, which you learned about in Chapter 9.

Reviewing Passing Variables *by Value*

Recall that, unless you specify otherwise, variables are passed *by value* in C++. This means that the computer passes only a copy of the variable's contents to the receiving function. When only a copy of the contents is passed, the receiving function is not given access to the variable in memory. Therefore, it cannot change the value stored inside of the variable. It is appropriate to pass a variable *by value* when the receiving function needs to *know* the variable's contents, but it does not need to *change* the contents. To illustrate, consider the C++ program shown in Figure 10-5. The program defines and calls a void function named `displayAge`. The `displayAge` function definition is located below the `main` function (on Lines 24 through 27). Therefore, the program includes an appropriate function prototype above the `main` function (on Line 8). Because the `displayAge` function is a void function, its function call (on Line 17) appears as a statement by itself. The function call passes the `age` variable *by value* to the `displayAge` function. Notice that the data type of the actual argument in the function call—in this case, `int`—matches the data type of the formal parameter listed in both the function header and function prototype. Also notice that the name of the actual argument (`age`) does not need to be identical to the name of the formal parameter (`years`). In fact, to avoid confusion, it is better to use different names for an actual argument and its corresponding formal parameter.



The receiving function can change only its copy of a variable passed *by value*. Changing the copy does not change the contents of the original variable.

```
 1  //Age.cpp - displays the user's age in a message
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  using namespace std;
 6
 7  //function prototype
 8  void displayAge(int years);
 9
10  int main()
11  {
12      int age = 0;
13      //get age
14      cout << "How old are you? ";
15      cin >> age;
16      //display age
17      displayAge(age);
18
19      system("pause");
20      return 0;
21  }   //end of main function
22
23  //*****function definitions*****
24  void displayAge(int years)
25  {
26      cout << "You are " << years << " years old." << endl;
27  }   //end of displayAge function
```

the name is not required in the function prototype

function call

your C++ development tool may not require this statement

function header

Figure 10-5 Age message program

To review the concept of passing *by value*, you will desk-check the program shown in Figure 10-5, using the number 34 as the age. The first statement in the `main` function creates and initializes an `int` variable named `age`. The variable is local to the `main` function and will remain in memory until the `main` function ends. The next two statements prompt the user to enter an age and then store the user's response (34) in the `age` variable. Figure 10-6 shows the desk-check table after the first three statements in the `main` function are processed.



Figure 10-6 Desk-check table after the first three statements in the `main` function are processed



Recall that the variables listed in a function header are local to the function and can be used only within the function. The variables will remain in memory until the function ends.

Next, the `displayAge(age);` statement on Line 17 calls the `displayAge` function, passing it a copy of the value stored in the `age` variable. In this case, the statement passes the integer 34. At this point, the computer temporarily leaves the `main` function to process the code contained in the `displayAge` function, beginning with the function header. The `displayAge` function header indicates that the computer should create one local `int` variable named `years`. The computer stores the value passed to the function in the `years` variable, as shown in Figure 10-7.

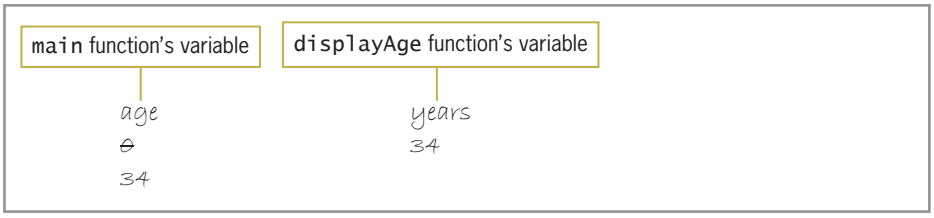


Figure 10-7 Desk-check table after the `displayAge` function header is processed

Next, the computer processes the `cout` statement contained in the `displayAge` function body. The statement displays string literal constants along with the contents of the function's local `years` variable. In this case, the statement will display the message "You are 34 years old." on the computer screen. The `displayAge` function ends when the computer encounters the function's closing brace. At that point, the computer removes the `years` variable from its internal memory. It then returns to the `main` function to process the statement immediately following the one that called the `displayAge` function. In this case, the `system("pause");` statement on Line 19 is the first statement immediately following the function call. Figure 10-8 shows the desk-check table after the `displayAge` function ends. Only the `main` function's local `age` variable remains in the computer's memory.

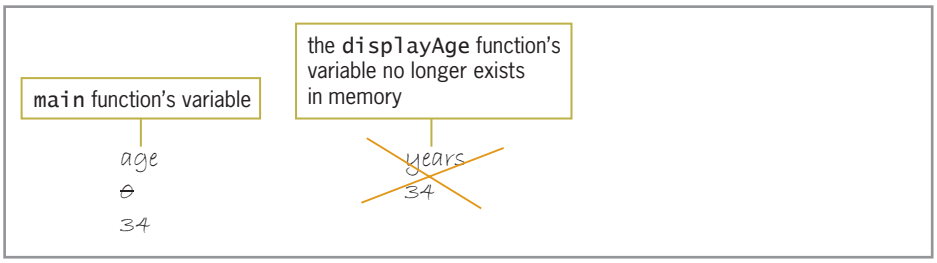


Figure 10-8 Desk-check table after the `displayAge` function ends

The `system("pause");` statement is processed next and pauses program execution until the user presses a key. After the user presses a key, the computer processes the `return 0;` statement on Line 20 in the `main` function. The statement returns the number 0 to the operating system to indicate that the program ended normally, and then the program ends. At this point, the computer removes the `main` function's local `age` variable from its internal memory. Figure 10-9 shows a sample run of the age message program.

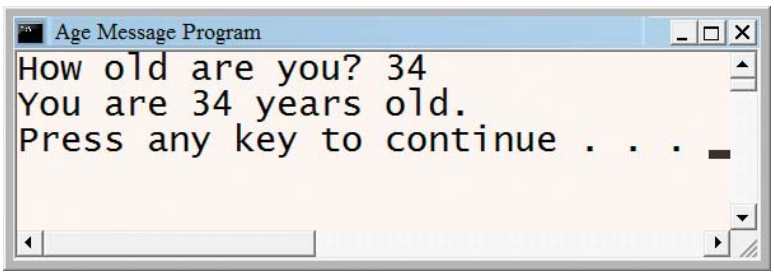


Figure 10-9 Sample run of the age message program

Passing Variables *by Reference*

Instead of passing a variable's value to a function, you can pass its address. In other words, you can pass the variable's location in the computer's internal memory. As you learned earlier, passing a variable's address is referred to as passing *by reference*, and it gives the receiving function access to the variable being passed. You pass a variable *by reference* when you want the receiving function to change the contents of the variable. To pass a variable *by reference* in C++, you include an ampersand (&) before the name of the corresponding formal parameter in the receiving function's header. The **&** (ampersand) is called the **address-of operator**, and it tells the computer to pass the variable's address rather than a copy of its contents. If the receiving function's definition appears below the `main` function in the program, you also must include the address-of operator in the receiving function's prototype. You enter the address-of operator immediately preceding the formal parameter's name, just as you do in the function header. If the prototype does not include the formal parameter's name, you enter a space followed by the address-of operator after the formal parameter's data type. We'll use a modified version of the age message program (shown earlier in Figure 10-5) to illustrate the concept of passing *by reference*. The modified program is shown in Figure 10-10, with the modifications shaded in the figure. In addition to

the `displayAge` function included in the original program, the modified program defines and calls a void function named `getAge`. The function call, which appears on Line 15, passes the `age` variable *by reference*. You can tell that the variable is passed *by reference* because the address-of operator precedes the formal parameter's name in the `getAge` function's header (on Line 24) and in its prototype (on Line 8). Notice that the statement that calls a function does not indicate whether an item is passed *by value* or *by reference*. You can determine that information only by examining the *parameterList* in either the receiving function's header or its prototype. Like the `displayAge` function, the `getAge` function is a void function; therefore, its function call appears as a statement by itself. Here again, notice that the data type of the actual argument in the function call matches the data type of the formal parameter listed in both the function header and function prototype. Also notice that the name of the actual argument (`age`) is different from the name of the formal parameter (`years`).

```
1  //Modified Age.cpp - displays the user's age in a message
2  //Created/revised by <your name> on <current date>
3
4  #include <iostream>
5  using namespace std;
6
7  //function prototypes
8  void getAge(int &years);
9  void displayAge(int years);
10
11 int main()
12 {
13     int age = 0;
14     //get age
15     getAge(age);
16     //display age
17     displayAge(age);
18
19     system("pause");
20     return 0;
21 }   //end of main function
22
23 //*****function definitions*****
24 void getAge(int &years)
25 {
26     cout << "How old are you? ";
27     cin >> years;
28 }   //end of getAge function
29
30 void displayAge(int years)
31 {
32     cout << "You are " << years << " years old." << endl;
33 }   //end of displayAge function
```

address-of operator

you also can use `void getAge(int &);`

address-of operator

Figure 10-10 Modified age message program

To help you understand the concept of passing *by reference*, you will desk-check the modified program shown in Figure 10-10, using the number 21 as the age. The first statement in the `main` function creates and initializes an `int` variable named `age`. The variable is local to the `main` function and will remain in memory until the `main` function ends. Figure 10-11 shows the desk-check table after the declaration statement is processed.



Figure 10-11 Desk-check table after the declaration statement in the `main` function is processed

Next, the `getAge(age);` statement on Line 15 calls the `getAge` function, passing it the address of the `age` variable. At this point, the computer temporarily leaves the `main` function to process the code contained in the `getAge` function, beginning with the function header. The formal parameter in the `getAge` function header indicates that the function receives the address of an `int` variable. When you pass a variable's address to a function, the computer uses the address to locate the variable in its internal memory. It then assigns the formal parameter's name to the memory location. In this case, the computer locates the `age` variable in memory and assigns the name `years` to it. As indicated in the desk-check table in Figure 10-12, the memory location now has two names: one assigned by the `main` function and the other assigned by the `getAge` function. Although both functions can access the memory location, each function uses a different name to do so. The `main` function uses the name `age`, whereas the `getAge` function uses the name `years`.



Figure 10-12 Desk-check table after the `getAge` function header is processed

Next, the computer processes the two statements contained in the `getAge` function body. The statements prompt the user to enter an age and then store the user's response (21) in the `years` variable. Figure 10-13 shows the desk-check table after the statements in the `getAge` function are processed. Notice that changing the value in the `years` variable also changes the value in the `age` variable. This is because both variable names refer to the same location in memory.



Figure 10-13 Desk-check table after the statements in the `getAge` function are processed

The `getAge` function ends when the computer encounters the function's closing brace. At that point, the computer removes the name of the function's formal parameter (`years`) from the appropriate location in memory. Figure 10-14 shows the desk-check table after the `getAge` function ends.



Figure 10-14 Desk-check table after the `getAge` function ends

After the `getAge` function ends, the computer returns to the `main` function to process the statement immediately following the one that called the `getAge` function. In this case, the computer will process the `displayAge(age);` statement on Line 17. The statement calls the `displayAge` function, passing it a copy of the value (21) stored in the `age` variable. You can tell that the `age` variable is passed *by value*, because there is no ampersand before the name of the formal parameter in the `displayAge` function's header and prototype. The computer temporarily leaves the `main` function to process the code contained in the `displayAge` function, beginning with the function header. The `displayAge` function header indicates that the computer should create one local `int` variable named `years`. The computer stores the value passed to the function in the `years` variable, as shown in Figure 10-15.

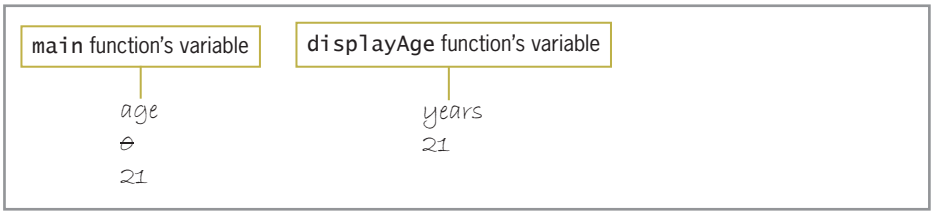


Figure 10-15 Desk-check table after the computer processes the `displayAge` function header

Next, the computer processes the `cout` statement contained in the `displayAge` function body. The statement will display the message "You are 21 years old." on the computer screen. When the computer encounters the `displayAge` function's closing brace, the function ends. The computer then removes the `years` variable from its internal memory. It then returns to the `main` function to process the `system("pause");` statement on Line 19. Figure 10-16 shows the desk-check table after the `displayAge` function ends. Only the `main` function's local `age` variable remains in the computer's memory.

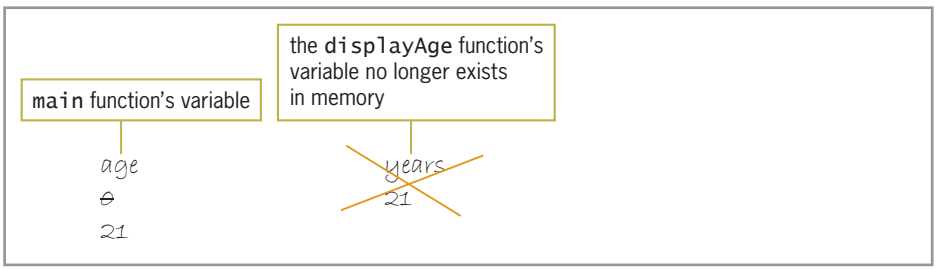


Figure 10-16 Status of the desk-check table after the `displayAge` function ends

The `system("pause");` statement pauses program execution until the user presses a key, after which the computer processes the `return 0;` statement on Line 20. The `return` statement returns the number 0 to the operating system to indicate that the program ended normally. When the program ends, the computer removes the `main` function's local `age` variable from its internal memory. A sample run of the modified age message program is shown in Figure 10-17.

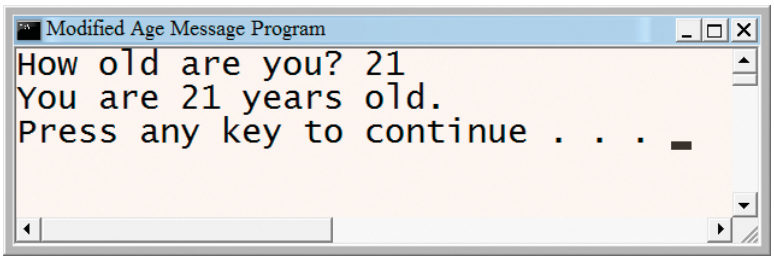


Figure 10-17 Sample run of the modified age message program

Keep in mind that when you pass a variable *by value*, the computer uses the data type and name of its corresponding formal parameter to create a separate memory location in which to store the value. When you pass a variable *by reference*, on the other hand, the computer locates the variable in memory and then assigns the name of its corresponding formal parameter to the memory location. When you pass a variable *by reference*, the variable will have two names: one assigned by the calling function and the other assigned by the receiving function. Void functions use variables that are passed *by reference* to send information back to the calling function. Value-returning functions, on the other hand, use their return value to send information back to the calling function.



Only variables can be passed *by reference*.



The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 10-2

1. Which of the following is a valid function header for a void function named `getInput`? The function is passed the addresses of two `double` variables.
   a. `void getInput(double &hours, double &rate)`
   b. `void getInput(&double hours, &double rate)`
   c. `void getInput(double hours, double rate)`
   d. `void getInput(double %hours, double %rate)`

2. Which of the following statements can be used to call the `getInput` function from Question 1, passing it the addresses of two `double` variables named `hoursWkd` and `payRate`?
   a. `getInput(&hoursWkd, &payRate);`
   b. `getInput(hoursWkd, payRate);`
   c. `getInput(payRate &, hoursWkd &);`
   d. `getInput(double &hoursWkd, double &payRate);`

3. Write the function prototype for the `getInput` function from Question 1.

4. When a variable is passed *by reference*, the computer assigns the name of its corresponding formal parameter to the same location in memory.
   a. True
   b. False

The Salary Program

In the modified age message program (shown earlier in Figure 10-10), each function call passed one variable to its respective function. The `getAge(age);` statement passed the `age` variable *by reference* to the `getAge` function, while the `displayAge(age);` statement passed the `age` variable *by value* to the `displayAge` function. The program you will view in this section contains a function call that passes more than one variable to a function. More specifically, it passes four variables: two *by value* and two *by reference*. Figure 10-18 shows the problem specification for the salary program. It also includes the IPO chart information and C++ instructions for the program's `main` and `getNewPayInfo` functions. The program allows the user to enter an employee's current salary and raise rate. It then calculates and displays the employee's raise and new salary. Notice that the `main` function calls the `getNewPayInfo` function to calculate the raise and new salary amounts. The `main` function passes four items of information to

the `getNewPayInfo` function: the value of the current salary, the value of the raise rate, the address of the variable where the raise can be stored after it is calculated, and the address of the variable where the new salary can be stored after it is calculated. In other words, the `main` function passes its `currentSalary` and `raiseRate` variables *by value*, and it passes its `raise` and `newSalary` variables *by reference*.

Problem specification
Create a program that allows the user to enter an employee's current salary and raise rate. The program should calculate and display the employee's raise and new salary.

`main` function

IPO chart information	**C++ instructions**
Input	
current salary	`double currentSalary = 0.0;`
raise rate	`double raiseRate = 0.0;`
Processing	
none	
Output	
raise	`double raise = 0.0;`
new salary	`double newSalary = 0.0;`
Algorithm	

1. enter the current salary and raise rate	`cout << "Current salary: ";`
	`cin >> currentSalary;`
	`cout << "Raise rate (in decimal form): ";`
	`cin >> raiseRate;`

2. call the getNewPayInfo function to calculate the raise and new salary; pass the function the current salary and raise rate, as well as the addresses of variables in which to store the raise and new salary	`getNewPayInfo(currentSalary, raiseRate, raise, newSalary);`
3. display the raise and new salary	`cout << "Raise: $" << raise << endl;` `cout << "New salary: $" << newSalary << endl;`

`getNewPayInfo` function

IPO chart information	**C++ instructions**
Input	
current salary (formal parameter)	`double current`
raise rate (formal parameter)	`double rate`
addresses of variables to store:	
raise (formal parameter)	`double &increase`
new salary (formal parameter)	`double &pay`

Figure 10-18 Problem specification, IPO chart information, and C++ instructions for the salary program *(continues)*

(continued)

Processing

none

Output

raise	stored in the increase formal parameter
new salary	stored in the pay formal parameter

Algorithm

1. calculate the raise by multiplying the current salary by the raise rate	`increase = current * rate;`
2. calculate the new salary by adding the raise to the current salary	`pay = current + increase;`

Figure 10-18 Problem specification, IPO chart information, and C++ instructions for the salary program

Figure 10-19 shows the C++ code for the entire salary program. The `getNewPayInfo` function's prototype, call, and header are shaded in the figure. Recall that the names of the formal parameters are optional in a function prototype. Therefore, you also could write the function prototype in Figure 10-19 as follows: `void getNewPayInfo(double, double, double &, double &);`. Figure 10-20 shows a sample run of the salary program.

```
 1  //Salary.cpp - displays the raise and new salary
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <iomanip>
 6  using namespace std;
 7
 8  //function prototype
 9  void getNewPayInfo(double current, double rate,
10                     double &increase, double &pay);
11
12  int main()
13  {
14      //declare variables
15      double currentSalary = 0.0;
16      double raiseRate     = 0.0;
17      double raise         = 0.0;
18      double newSalary     = 0.0;
19
20      //get input items
21      cout << "Current salary: ";
22      cin >> currentSalary;
23      cout << "Raise rate (in decimal form): ";
24      cin >> raiseRate;
25
```

Figure 10-19 Salary program *(continues)*

(continued)

```
26      //get the raise and new salary
27      getNewPayInfo(currentSalary, raiseRate,
28                    raise, newSalary);
29
30      //display the raise and new salary
31      cout << fixed << setprecision(2);
32      cout << "Raise: $" << raise << endl;
33      cout << "New salary: $" << newSalary << endl;
34
35      system("pause");
36      return 0;
37  }   //end of main function
38
39  //*****function definitions*****
40  void getNewPayInfo(double current, double rate,
41                     double &increase, double &pay)
42  {
43      increase = current * rate;
44      pay = current + increase;
45  }   //end of getNewPayInfo function
```

Figure 10-19 Salary program

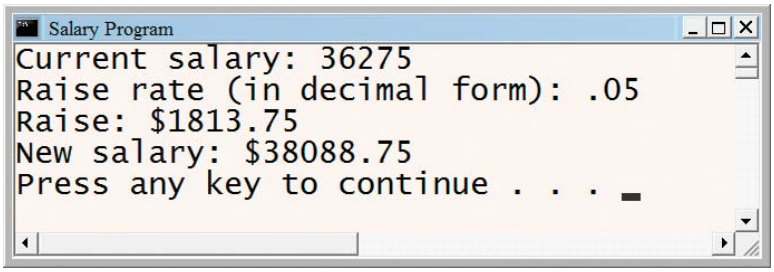

Salary Program
Current salary: 36275
Raise rate (in decimal form): .05
Raise: $1813.75
New salary: $38088.75
Press any key to continue . . .

Figure 10-20 Sample run of the salary program

You will desk-check the salary program using $36275 and .05 as the current salary and raise rate, respectively. The first four statements in the `main` function create and initialize four `double` variables named `currentSalary`, `raiseRate`, `raise`, and `newSalary`. The next four statements prompt the user to enter the current salary and raise rate, storing the user's responses in the `currentSalary` and `raiseRate` variables. Figure 10-21 shows the desk-check table after the computer processes the first eight statements in the `main` function.

currentSalary (main)	raiseRate (main)	raise (main)	newSalary (main)
~~0.0~~	~~0.0~~	0.0	0.0
36275.0	.05		

Figure 10-21 Desk-check table after the statements on Lines 15 through 24 are processed

Next, the computer processes the `getNewPayInfo(currentSalary, raiseRate, raise, newSalary);` statement. The statement calls the `getNewPayInfo` function, passing it four variables. The `currentSalary` and `raiseRate` variables are passed *by value*, because the `getNewPayInfo` function needs to *know* the values stored in the variables, but it does not need to *change* those values. You can tell that the variables are passed *by value*, because their corresponding formal parameters (which are listed in the function header and function prototype) do not contain an ampersand. The `raise` and `newSalary` variables, on the other hand, are passed *by reference*, as the ampersand in their formal parameters indicates. It is necessary to pass the `raise` and `newSalary` variables *by reference*, because the `getNewPayInfo` function needs to store the raise and new salary amounts in those memory locations after the amounts are calculated. At this point, the computer temporarily leaves the `main` function to process the code contained in the `getNewPayInfo` function, beginning with the function header. When processing the `double current, double rate` portion of the function header, the computer creates two `double` memory locations named `current` and `rate`. It stores the value of the `currentSalary` variable (36275.0) in the `current` memory location and stores the value of the `raiseRate` variable (.05) in the `rate` memory location. When processing the `double &increase, double &pay` portion of the function header, the computer assigns the name `increase` to the `main` function's `raise` variable and assigns the name `pay` to the `main` function's `newSalary` variable. Figure 10-22 shows the desk-check table after the computer processes the `getNewPayInfo` function header.

currentSalary (main)	raiseRate (main)	increase (getNewPayInfo) raise (main)	pay (getNewPayInfo) newSalary (main)
~~0.0~~ 36275.0	~~0.0~~ .05	0.0	0.0

current (getNewPayInfo)	rate (getNewPayInfo)
36275.0	.05

Figure 10-22 Desk-check table after the computer processes the `getNewPayInfo` function header

The statements contained in the `getNewPayInfo` function body are processed next. The first statement multiplies the contents of the `current` variable by the contents of the `rate` variable and then assigns the result (1813.75) to the `increase` variable. The second statement adds together the contents of the `current` and `increase` variables and then assigns the sum (38088.75) to the `pay` variable. Figure 10-23 shows the desk-check table after the computer processes the statements in the `getNewPayInfo` function body.

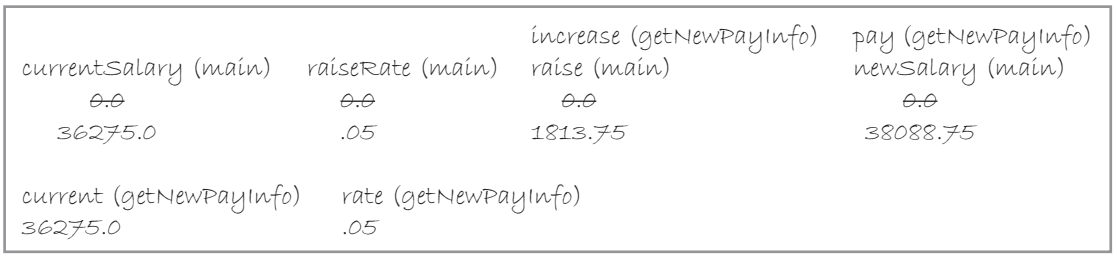


Figure 10-23 Desk-check table after the computer processes the statements in the `getNewPayInfo` function body

When the computer encounters the `getNewPayInfo` function's closing brace, the function ends. The computer then removes the `increase` and `pay` names from the appropriate locations in memory. It also removes the `getNewPayInfo` function's `current` and `rate` variables from memory. Figure 10-24 shows the desk-check table after the `getNewPayInfo` function ends. Notice that only the `main` function's variables remain in the computer's memory.

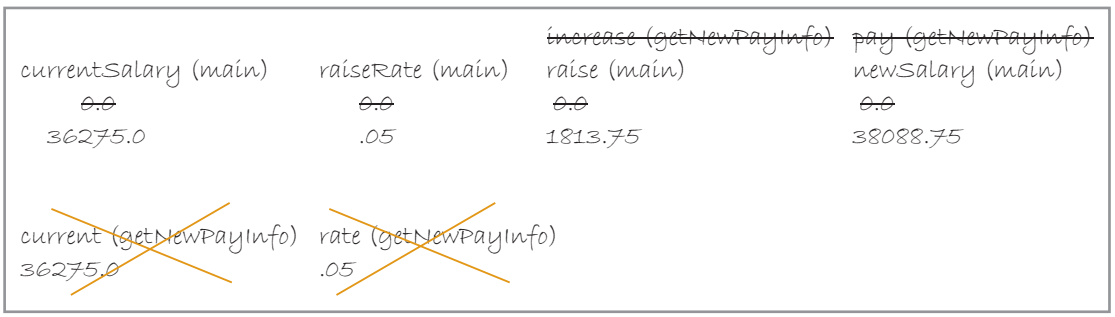


Figure 10-24 Desk-check table after the `getNewPayInfo` function ends

After the `getNewPayInfo` function ends, the computer returns to the `main` function and processes the statement immediately following the function call. In this case, it processes the `cout << fixed << setprecision(2);` statement on Line 31. The statement tells the computer to display real numbers using fixed-point notation with two decimal places. Next, the computer processes the `cout` statements on Lines 32 and 33. Those statements display the raise and new salary amounts on the screen. The `system("pause");` statement on Line 35 is processed next, followed by the `return 0;` statement on Line 36. When the `main` function ends, the computer removes the function's local variables (`currentSalary`, `raiseRate`, `raise`, and `newSalary`) from memory.

Mini-Quiz 10-3



The answers to Mini-Quiz questions are located in Appendix A.

1. Write the function header for a void function named `calcTaxes`. The function is passed the value of a `double` variable named `gross` and the addresses of two `double` variables named `federal` and `state`. Use `pay`, `fedTax`, and `stateTax` for the names of the formal parameters.

2. Write a C++ statement to call the `calcTaxes` function from Question 1?

3. Write the function prototype for the `calcTaxes` function from Question 1.

4. Unless specified otherwise, a variable's address is passed to a function in C++.

   a. True

   b. False



The answers to the labs are located in Appendix A.



LAB 10-1 Stop and Analyze

Figure 10-25 shows a sample run of the program for Lab 10-1, and Figure 10-26 shows the program's code. Study the program shown in Figure 10-26, and then answer the questions.

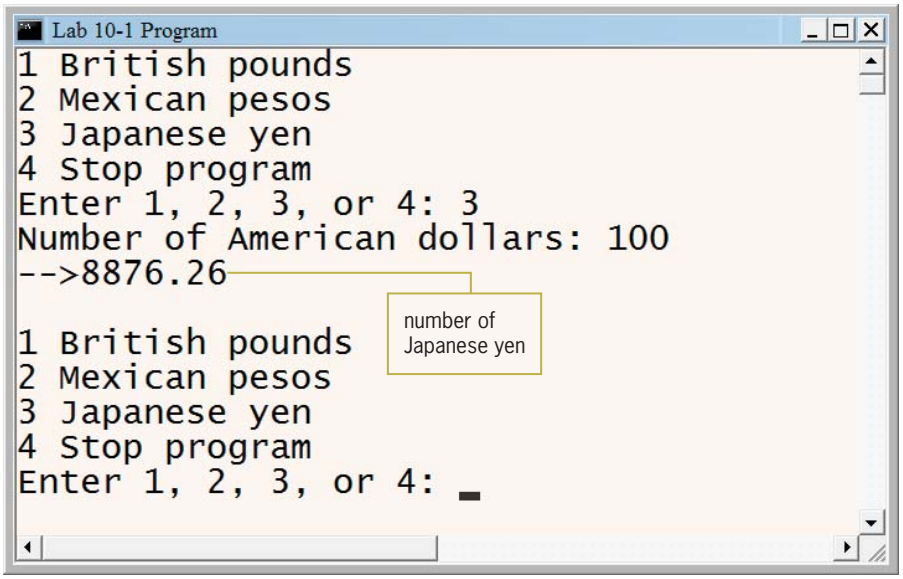


Figure 10-25 Sample run of the program for Lab 10-1

```
 1  //Lab10-1.cpp - Converts American dollars to
 2  //British pounds, Mexican pesos, or Japanese yen
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  #include <iomanip>
 7  using namespace std;
 8
 9  //function prototypes
10  void displayMenu();
```

Figure 10-26 Code for Lab 10-1 (continues)

(continued)

```
11  void convertDols(double dollars,
12                   double convertRate,
13                   double &converted);
14
15  int main()
16  {
17      //declare constants
18      const double BRITISH_RATE  = .630676;
19      const double MEXICAN_RATE  = 13.5584;
20      const double JAPANESE_RATE = 88.7626;
21
22      //declare variables
23      int menuChoice           = 0;
24      double americanDollars   = 0.0;
25      double conversionRate    = 0.0;
26      double convertedCurrency = 0.0;
27
28      //display output in fixed-point notation
29      //with two decimal places
30      cout << fixed << setprecision(2);
31
32      //get menu choice
33      displayMenu();
34      cout << "Enter 1, 2, 3, or 4: ";
35      cin >> menuChoice;
36
37      while (menuChoice > 0 && menuChoice < 4)
38      {
39           //get dollars to convert
40           cout << "Number of American dollars: ";
41           cin >> americanDollars;
42
43           //assign rate
44           if (menuChoice == 1)
45                conversionRate = BRITISH_RATE;
46           else if (menuChoice == 2)
47                conversionRate = MEXICAN_RATE;
48           else
49                conversionRate = JAPANESE_RATE;
50           //end if
51
52           convertDols(americanDollars,
53                       conversionRate,
54                       convertedCurrency);
55           cout << "-->" << convertedCurrency
56                << endl << endl;
57
58           //get menu choice
59           displayMenu();
60           cout << "Enter 1, 2, 3, or 4: ";
61           cin >> menuChoice;
62      }    //end while
```

Figure 10-26 Code for Lab 10-1 (continues)

(continued)

```
63
64      system("pause");
65      return 0;
66  }   //end of main function
67
68  //*****function definitions*****
69  void displayMenu()
70  {
71      cout << "1 British pounds" << endl;
72      cout << "2 Mexican pesos" << endl;
73      cout << "3 Japanese yen" << endl;
74      cout << "4 Stop program" << endl;
75  }   //end of displayMenu function
76
77  void convertDols(double dollars,
78                   double convertRate,
79                   double &converted)
80  {
81      converted = dollars * convertRate;
82  }   //end of convertDols function
```

Figure 10-26 Code for Lab 10-1

QUESTIONS

1. The `main` function passes three variables to the `convertDols` function. Which lines in the program indicate whether the variables are passed *by value* or *by reference*?
2. Why are the `americanDols` and `conversionRate` variables passed *by value*? Why is the `convertedCurrency` variable passed *by reference*?
3. Why is the `displayMenu` function a void function?
4. Which line in the program contains the priming read? Which line contains the update read?
5. The `convertDols` function in Figure 10-26 is a void function. How would you modify the function to make it a value-returning function? What other changes would you need to make to the program?
6. Follow the instructions for starting C++ and opening the Lab10-1.cpp file. The file is contained in either the Cpp6\Chap10\Lab10-1 Project folder or the Cpp6\Chap10 folder. Run the program. First, you will use the program to convert 100 American dollars to Japanese yen. Type 3 and press Enter, and then type 100 and press Enter. The program displays the number 8876.26, as shown earlier in Figure 10-25.
7. Use the program to convert 50 American dollars to British pounds. What is the answer?
8. Use the program to convert 10 American dollars to Mexican pesos. What is the answer?

9. Modify the program so that it uses a void function to assign the conversion rate. Name the function `assignRate`. Save and then run the program. Test the program using the data from Steps 6, 7, and 8.
10. Now, change the `convertDols` function to a value-returning function. (Refer to Step 5.) Save and then run the program. Test the program using the data from Steps 6, 7, and 8.



LAB 10-2 Plan and Create

In this lab, you will plan and create an algorithm for Addison Clarke. The problem specification and a sample calculation are shown in Figure 10-27.

Addison Clarke works for her local electric company. She wants a program that calculates a customer's electric bill. Addison will enter the current and previous meter readings. The program should calculate and display the number of units of electricity used and the total charge for the electricity. The charge for each unit of electricity is $0.09.

Example

Current reading:	53512
Previous reading:	51875
Units used:	1637
Charge per unit:	* .09
Total charge:	$147.33

Figure 10-27 Problem specification and a sample calculation for Lab 10-2

First, analyze the problem, looking for the output first and then for the input. In this case, Addison wants the program to display the number of units of electricity used and the total charge for the electricity. To calculate these amounts, the computer will need to know the previous meter reading, the current meter reading, and the charge per unit of electricity. Next, plan the algorithm. Recall that most algorithms begin with an instruction to enter the input items into the computer, followed by instructions that process the input items, typically including the items in one or more calculations. Most algorithms end with one or more instructions that display, print, or store the output items. In this program, you will use three program-defined void functions named `getInput`, `calcBill`, and `displayBill`. Each void function will be called by the program's `main` function. The `getInput` function will get the current and previous meter readings from the user. A void function is appropriate in this case because the `getInput` function needs to get two values for the `main` function, and a value-returning function can return only one value. The `calcBill` function will be responsible for calculating both the number of units used and the total charge. Here again, a void function is appropriate because the `calcBill` function needs to calculate more than one value for the `main` function. The `displayBill` function will perform the task of displaying both the number of units used and the total charge. The

`displayBill` function should be a void function because it will not need to return a value to the `main` function after completing its task. Figure 10-28 shows the IPO charts for the program's `main`, `getInput`, `calcBill`, and `displayBill` functions. Notice that the `getInput` function will be passed the addresses of variables that it can use to store the current and previous meter readings. The `calcBill` function will be passed five items: the current reading, the previous reading, the unit charge, the address of a variable that it can use to store the number of units used, and the address of a variable that it can use to store the total charge. The first three items are necessary to calculate the number of units used and the total charge. The last two items provide memory locations in which to store the calculated results. The `displayBill` function will be passed two items: the number of units used and the total charge.

`main` function

Input	Processing	Output
current reading previous reading unit charge (.09)	Processing items: none Algorithm: 1. call the getInput function to get the current reading and previous reading, pass the function the addresses of variables in which to store the current reading and previous reading 2. call the calcBill function to calculate the units used and total charge, pass the function the current reading, previous reading, and unit charge, as well as the addresses of variables in which to store the units used and total charge 3. call the displayBill function to display the units used and total charge, pass the function the units used and total charge	units used total charge

`getInput` function

Input	Processing	Output
addresses of variables to store: current reading previous reading	Processing items: none Algorithm: 1. enter the current reading 2. enter the previous reading	current reading previous reading

Figure 10-28 IPO charts for the functions in the electric bill program *(continues)*

(continued)

calcBill function

Input	Processing	Output
current reading previous reading unit charge addresses of variables to store: units used total charge	Processing items: none Algorithm: 1. calculate the units used by subtracting the previous reading from the current reading 2. calculate the total charge by multiplying the units used by the unit charge	units used total charge

displayBill function

Input	Processing	Output
units used total charge	Processing items: none Algorithm: 1. display the units used 2. display the total charge	units used total charge

Figure 10-28 IPO charts for the functions in the electric bill program

After completing the IPO charts, you then move on to the third step in the problem-solving process, which is to desk-check the algorithm. You will desk-check the algorithm in Figure 10-28 using 53512 and 51875 as the current and previous meter readings. Using these values, the number of units used and total charge should be 1637 and $147.33, respectively. Figure 10-29 shows the completed desk-check table.

~~current reading (getInput)~~ current reading (main) 53512	~~previous reading (getInput)~~ previous reading (main) 51875	unit charge (main) .09
~~units used (calcBill)~~ units used (main) 1637	~~total charge (calcBill)~~ total charge (main) 147.33	
~~current reading (calcBill)~~ 53512	~~previous reading (calcBill)~~ 51875	~~unit charge (calcBill)~~ .09
~~units used (displayBill)~~ 1637	~~total charge (displayBill)~~ 147.33	

Figure 10-29 Completed desk-check table for the electric bill algorithms

The fourth step in the problem-solving process is to code the algorithm into a program. The IPO chart information and C++ instructions for the electric bill program are shown in Figure 10-30.

`main` function

IPO chart information	**C++ instructions**
Input	
current reading	`int current = 0;`
previous reading	`int previous = 0;`
unit charge (.09)	`const double UNIT_CHG = .09;`
Processing	
none	
Output	
units used	`int units = 0;`
total charge	`double total = 0.0;`
Algorithm	
1. call the getInput function to get the current reading and previous reading, pass the function the addresses of variables in which to store the current reading and previous reading	`getInput(current, previous);`
2. call the calcBill function to calculate the units used and total charge, pass the function the current reading, previous reading, and unit charge, as well as the addresses of variables in which to store the units used and total charge	`calcBill(current, previous, UNIT_CHG, units, total);`
3. call the displayBill function to display the units used and total charge, pass the function the units used and total charge	`displayBill(units, total);`

`getInput` function

IPO chart information	**C++ instructions**
Input	
addresses of variables to store:	
current reading (formal parameter)	`int &newReading`
previous reading (formal parameter)	`int &oldReading`
Processing	
none	
Output	
current reading	stored in the newReading formal parameter
previous reading	stored in the oldReading formal parameter

Figure 10-30 IPO chart information and C++ instructions for the electric bill program *(continues)*

(continued)

Algorithm

1. enter the current reading	`cout << "Current reading: ";` `cin >> newReading;`
2. enter the previous reading	`cout << "Previous reading: ";` `cin >> oldReading;`

`calcBill` function

IPO chart information	C++ instructions
Input	
current reading (formal parameter)	`int curRead`
previous reading (formal parameter)	`int prevRead`
unit charge (formal parameter)	`double chgPerUnit`
addresses of variables to store:	
units used (formal parameter)	`int &numUnits`
total charge (formal parameter)	`double &totChg`
Processing	
none	
Output	
units used	stored in the numUnits formal parameter
total charge	stored in the totChg formal parameter

Algorithm

1. calculate the units used by subtracting the previous reading from the current reading	`numUnits = curRead - prevRead;`
2. calculate the total charge by multiplying the units used by the unit charge	`totChg = numUnits * chgPerUnit;`

`displayBill` function

IPO chart information	C++ instructions
Input	
units used (formal parameter)	`int used`
total charge (formal parameter)	`double charge`
Processing	
none	
Output	
units used	
total charge	

Algorithm

1. display the unit used	`cout << "Units used: " << used << endl;`
2. display the total charge	`cout << "Total charge: $" << charge << endl;`

Figure 10-30 IPO chart information and C++ instructions for the electric bill program

The fifth step in the problem-solving process is to desk-check the program. Figure 10-31 shows the entire electric bill program, and Figure 10-32 shows the completed desk-check table.

```
 1  //Lab10-2.cpp - displays the number of units of
 2  //electricity used and the total charge
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  #include <iomanip>
 7  using namespace std;
 8
 9  //function prototypes
10  void getInput(int &newReading, int &oldReading);
11  void calcBill(int curRead, int prevRead,
12                double chgPerUnit, int &numUnits,
13                double &totChg);
14  void displayBill(int used, double charge);
15
16  int main()
17  {
18      //declare constant and variables
19      const double UNIT_CHG = .09;
20      int current           = 0;
21      int previous          = 0;
22      int units             = 0;
23      double total          = 0.0;
24
25      cout << fixed << setprecision(2);
26
27      //call functions
28      getInput(current, previous);
29      calcBill(current, previous, UNIT_CHG, units, total);
30      displayBill(units, total);
31
32      system("pause");
33      return 0;
34  }   //end of main function
35
36  //*****function definitions*****
37  void getInput(int &newReading, int &oldReading)
38  {
39      cout << "Current reading: ";
40      cin >> newReading;
41      cout << "Previous reading: ";
42      cin >> oldReading;
43  }   //end of getInput function
44
45  void calcBill(int curRead, int prevRead,
46                double chgPerUnit, int &numUnits,
47                double &totChg)
48  {
49      numUnits = curRead - prevRead;
```

your C++ development tool may not require this statement

Figure 10-31 Electric bill program *(continues)*

(continued)

```
50      totChg = numUnits * chgPerUnit;
51  }   //end of calcBill function
52
53  void displayBill(int used, double charge)
54  {
55      cout << "Units used: " << used << endl;
56      cout << "Total charge: $" << charge << endl;
57  }   //end of displayBill function
```

Figure 10-31 Electric bill program

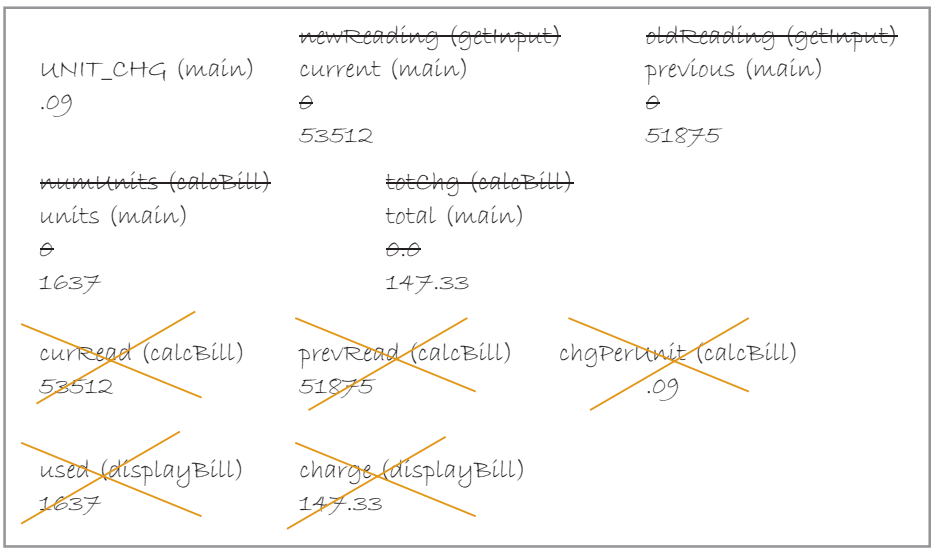


Figure 10-32 Completed desk-check table for the electric bill program

The final step in the problem-solving process is to evaluate and modify (if necessary) the program. Recall that you evaluate a program by entering its instructions into the computer and then using the computer to run (execute) it. While the program is running, you enter the same sample data used when desk-checking the program.

DIRECTIONS

Follow the instructions for starting your C++ development tool. Depending on the development tool you are using, you may need to create a new project; if so, name the project Lab10-2 Project and save it in the Cpp6\Chap10 folder. Enter the instructions shown in Figure 10-31 in a source file named Lab10-2.cpp. (Do not enter the line numbers.) Save the file in either the project folder or the Cpp6\Chap10 folder. Now, follow the appropriate instructions for running the Lab10-2.cpp file. Test the program using 53512 and 51875 as the current and previous meter readings. The number of units used and the total charge should be 1637 and $147.33, respectively. Also test the program using your own sample data. If necessary, correct any bugs (errors) in the program.



LAB 10-3 Modify

If necessary, create a new project named Lab10-3 Project. Enter (or copy) the Lab10-2.cpp instructions into a new source file named Lab10-3.cpp. Change Lab10-2.cpp in the first comment to Lab10-3.cpp. Currently, the electric bill program uses one void function to calculate both the number of units used and the total charge. Replace the `calcBill` function with two functions: a void function that calculates the number of units used and a value-returning function that calculates and returns the total charge. Name the functions `getUnits` and `getTotal`. Save and then run the program. Test the program appropriately.



LAB 10-4 Desk-Check

Desk-check the code in Figure 10-33 using the following four sets of test scores: 78 and 85, 45 and 93, 87 and 98, and 54 and 32.

```
 1  //Lab10-4.cpp - displays the average test score
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <iomanip>
 6  using namespace std;
 7
 8  //function prototype
 9  void calcAvg(double num1, double num2, double &avg);
10
11  int main()
12  {
13      //declare variables
14      double test1   = 0.0;
15      double test2   = 0.0;
16      double average = 0.0;
17
18      cout << "First test score ";
19      cout << "(negative number to end): ";
20      cin >> test1;
21      while (test1 >= 0)
22      {
23           cout << "Second test score: ";
24           cin >> test2;
25
26           //call function to calculate the average
27           calcAvg(test1, test2, average);
28
29           //display the average
30           cout << fixed << setprecision(1);
31           cout << "Average score: " << average
32                << endl << endl;
```

Figure 10-33 Code for Lab 10-4 *(continues)*

(continued)

```
33
34          cout << "First test score ";
35          cout << "(negative number to end): ";
36          cin >> test1;
37      }   //end while
38
39      system("pause");
40      return 0;
41  }   //end of main function
42
43  //*****function definitions*****
44  void calcAvg(double num1, double num2, double &avg)
45  {
46      avg = (num1 + num2) / 2;
47  }   //end of calcAvg function
```

Figure 10-33 Code for Lab 10-4



LAB 10-5 Debug

Follow the instructions for starting C++ and opening the Lab10-5.cpp file. The file is contained in either the Cpp6\Chap10\Lab10-5 Project folder or the Cpp6\Chap10 folder. Run the program. Enter the following scores: 93, 90, 85, and 100. The program should display 368 as the total points and A as the grade. Debug the program.

Summary

- All functions fall into one of two categories: value-returning or void. A value-returning function returns precisely one value to the statement that called the function. A void function, on the other hand, does not return a value.
- Like a value-returning function, a void function is composed of a function header and a function body. However, unlike a value-returning function, the function header for a void function begins with the keyword `void` rather than with a data type. Also unlike a value-returning function, the function body for a void function does not contain a `return` statement.
- As you do when calling a value-returning function, you call a void function by including its name and actual arguments (if any) in a statement. However, unlike a call to a value-returning function, a call to a void function appears as a statement by itself rather than as part of another statement. When the computer finishes processing a void function's code, it continues program execution with the statement immediately below the one that called the function.

- Variables can be passed to functions either *by value* (the default) or *by reference*.
- When you pass a variable *by value*, only a copy of the value stored inside of the variable is passed to the receiving function. The receiving function is not given access to a variable passed *by value*, so it cannot change the variable's contents.
- When you pass a variable *by value*, the computer uses the data type and name of the corresponding formal parameter to create a separate memory location in which to store a copy of the value.
- When you pass a variable *by reference*, the variable's address in memory is passed to the receiving function, allowing the receiving function to change the variable's contents. Only variables can be passed *by reference*.
- When you pass a variable *by reference*, the computer locates the variable in memory and then assigns the name of its corresponding formal parameter to the memory location. Therefore, the variable will have two names: one assigned by the calling function and the other assigned by the receiving function.
- To pass a variable *by reference* in a C++ program, you include the address-of operator (&) before the name of the corresponding formal parameter in the function header. If the function definition appears below the `main` function in the program, you also must include the address-of operator in the function prototype. The address-of operator tells the computer to pass the variable's address rather than its contents.

Key Terms

&—the address-of operator

Address-of operator—the ampersand; tells the computer to pass a variable's address in memory rather than its contents

Passing *by reference*—refers to the process of passing a variable's address to a function

Void functions—functions that do not return a value after completing their assigned task

Review Questions

1. Which of the following is false?
   a. A void function does not return a value after completing its assigned task.
   b. A void function call typically appears as its own statement in a C++ program.
   c. A void function cannot receive any items of information when it is called.
   d. A void function header begins with the keyword `void`.

2. Which of the following C++ statements correctly calls a void function named `displayTotal`, passing it an `int` variable named `total`?

   a. `cout << displayTotal(int total);`

   b. `cout << displayTotal(total);`

   c. `displayTotal(int total);`

   d. `displayTotal(total);`

3. A void function named `calcEndingBalance` is passed the values stored in two `int` variables. Which of the following function prototypes is correct for this function?

   a. `void calcEndingBalance(int, int);`

   b. `void calcEndingBalance(int, int)`

   c. `void calcEndingBalance(int &, int &);`

   d. `int calcEndingBalance(void);`

4. A void function named `calcEndingInventory` is passed four `int` variables named `beginInventory`, `sales`, `purchases`, and `endingInventory`. The function's task is to calculate the ending inventory, using the beginning inventory, sales, and purchase amounts passed to the function. The function should store the result in the `endingInventory` variable. Which of the following function headers is correct?

   a. `void calcEndingInventory(int b, int s, int p, int &e)`

   b. `void calcEndingInventory(int b, int s, int p, int e)`

   c. `void calcEndingInventory(int &b, int &s, int &p, int e)`

   d. `void calcEndingInventory(&int b, &int s, &int p, &int e)`

5. Which of the following statements calls the `calcEndingInventory` function described in Review Question 4?

   a. `calcEndingInventory(int, int, int, int);`

   b.
   ```
   calcEndingInventory(beginInventory, sales,
   purchases, &endingInventory);
   ```

   c.
   ```
   calcEndingInventory(beginInventory, sales,
   purchases, endingInventory);
   ```

   d.
   ```
   calcEndingInventory(int beginInventory, int
   sales, int purchases, int &endingInventory);
   ```

6. Unless specified otherwise, variables in C++ are passed _____.

   a. *by address*

   b. *by number*

   c. *by reference*

   d. *by value*

7. If you want the receiving function to change the contents of a variable, you must pass the variable _____.

   a. *by address*

   b. *by number*

   c. *by reference*

   d. *by value*

8. To determine whether an item is being passed *by value* or *by reference*, you must examine either the _____ or the _____.

   a. function call, function header

   b. function call, function prototype

   c. function header, function prototype

   d. function header, function body

9. Which of the following calls a void function named `displayName`, passing it no actual arguments?

   a. `call displayName();`

   b. `displayName;`

   c. `displayName()`

   d. `displayName();`

10. Which of the following is a correct function prototype for a void function that requires no formal parameters? The function's name is `displayName`.

    a. `displayName();`

    b. `void displayName;`

    c. `void displayName();`

    d. `void displayName(none);`

11. When a void function ends, the computer continues program execution with _____.

    a. the statement immediately above the one that called the function

    b. the statement that called the function

    c. the statement immediately below the one that called the function

12. If the function definitions section is located below the `main` function in a program, the program will have one function prototype for each program-defined function.

    a. True

    b. False

13. Variables that can be used only by the function in which they are declared are called _____ variables.

    a. global

    b. local

    c. separate

    d. void

14. Which of the following is false?

    a. When you pass a variable *by reference*, the receiving function can change the variable's contents.

    b. When you pass a variable *by value*, the receiving function creates a local variable that it uses to store the value.

    c. Unless specified otherwise, all variables in C++ are passed *by value*.

    d. To pass a variable *by reference* in C++, you place an ampersand (&) before the variable's name in the statement that calls the function.

15. A program contains a void function named `calcNewPrice`. The function receives two `double` variables named `oldPrice` and `newPrice`. The function multiplies the contents of the `oldPrice` variable by 1.1 and then stores the result in the `newPrice` variable. Which of the following is the appropriate function prototype for this function?

    a. `void calcNewPrice(double, double);`

    b. `void calcNewPrice(double &, double);`

    c. `void calcNewPrice(double, double &);`

    d. `void calcNewPrice(double &, double &);`

16. Which of the following can be used to call the `calcNewPrice` function described in Review Question 15?

    a. `calcNewPrice(double oldPrice, double newPrice);`

    b. `calcNewPrice(&oldPrice, newPrice);`

    c. `calcNewPrice(oldPrice, &newPrice);`

    d. `calcNewPrice(oldPrice, newPrice);`

17. Which of the following is false?

    a. The names of the formal parameters in the function header must be identical to the names of the actual arguments in the function call.

    b. When listing the formal parameters in a function header, you include each parameter's data type and name.

    c. The formal parameters should be the same data type as the actual arguments.

    d. If a function call passes an `int` variable first and a `char` variable second, the receiving function should receive an `int` variable followed by a `char` variable.

Exercises



Pencil and Paper

TRY THIS

1. Write the C++ code for a void function that receives an integer passed to it. The function should divide the integer by two and then display the result, which may contain a decimal place. Name the function `divideByTwo`. Name the formal parameter `wholeNumber`. (The answers to TRY THIS Exercises are located at the end of the chapter.)

TRY THIS

2. Write the function prototype for the `divideByTwo` function from Pencil and Paper Exercise 1. (The answers to TRY THIS Exercises are located at the end of the chapter.)

TRY THIS

3. Write a statement that calls the `divideByTwo` function from Pencil and Paper Exercise 1, passing the function the contents of the `total` variable. (The answers to TRY THIS Exercises are located at the end of the chapter.)

MODIFY THIS

4. Rewrite the code from Pencil and Paper Exercises 1, 2, and 3 so that the `divideByTwo` function receives two integers rather than one integer. The function should divide the sum of both integers by two and then display the result. Name the formal parameters `num1` and `num2`. Name the actual arguments `total1` and `total2`.

INTRODUCTORY

5. Write the C++ code for a void function that receives three `double` variables: the first two *by value* and the last one *by reference*. Name the formal parameters `n1`, `n2`, and `answer`. The function should divide the `n1` variable by the `n2` variable and then store the result in the `answer` variable. Name the function `calcQuotient`. Also write an appropriate function prototype for the `calcQuotient` function. In addition, write a statement that invokes the `calcQuotient` function, passing it the `num1`, `num2`, and `quotient` variables.

INTERMEDIATE

6. Write the C++ code for a void function that receives four `int` variables: the first two *by value* and the last two *by reference*. Name the formal parameters `n1`, `n2`, `sum`, and `diff`. The function should calculate the sum of the two variables passed *by value* and then store the result in the first variable passed *by reference*. It also should calculate the difference between the two variables passed *by value* and then store the result in the second variable passed *by reference*. When calculating the difference, subtract the contents of the `n2` variable from the contents of the `n1` variable. Name the function `calcSumAndDiff`. Also write an appropriate function prototype for the `calcSumAndDiff` function. In addition, write a statement that invokes the `calcSumAndDiff` function, passing it the `num1`, `num2`, `numSum`, and `numDiff` variables.

7. Write the C++ code for a function that receives five `double` numbers: four *by value* and one *by reference*. Name the formal parameters `num1`, `num2`, `num3`, `num4`, and `avg`. The function should calculate the average of the four numbers and then assign the result to the `avg` variable. Name the function `calcAverage`. Also write an appropriate function prototype for the `calcAverage` function. In addition, write a statement that invokes the `calcAverage` function. Use the following numbers and variable as the actual arguments: 45.67, 8.35, 125.78, 99.56, and `numAvg`.

INTERMEDIATE

8. Desk-check the program shown in Figure 10-34. Show the desk-check table after the first four statements in the `main` function are processed. Also show it after the statement in the `calcEnd` function is processed. Finally, show the desk-check table after the `calcEnd` function ends.

INTERMEDIATE

```
1  //Fig10-34.cpp - displays the ending balance
2  //Created/revised by <your name> on <current date>
3
4  #include <iostream>
5  using namespace std;
6
7  void calcEnd(int beg, int pur,
8               int sale, int &ending);
9
10 int main()
11 {
12     int begVal   = 1000;
13     int purchase = 500;
14     int sale     = 200;
15     int endVal   = 0;
16
17     calcEnd(begVal, purchase, sale, endVal);
18
19     cout << "Ending value: " << endVal << endl;
20
21     system("pause");
22     return 0;
23 }   //end of main function
24
25 //*****function definitions*****
26 void calcEnd(int beg, int pur,
27              int sale, int &ending)
28 {
29     ending = beg + pur - sale;
30 }   //end of calcEnd function
```

Figure 10-34

9. A program's `main` function declares three `double` variables named `sales`, `taxRate`, and `salesTax`. The `main` function contains the following statement: `calcSalesTax(sales, taxRate, salesTax);`. The `calcSalesTax` function is responsible for calculating the sales tax. Its function header looks like this: `void calcSalesTax(double sold, double rate, double tax)`. Correct the function header.

SWAT THE BUGS



Computer

TRY THIS

10. In this exercise, you experiment with passing variables *by value* and *by reference*. (The answers to TRY THIS Exercises are located at the end of the chapter.)

    a. Follow the instructions for starting C++ and opening the TryThis10.cpp file. The file is contained in either the Cpp6\Chap10\TryThis10 Project folder or the Cpp6\Chap10 folder. Notice that the `main` function passes the `age` variable *by value* to the `getAge` function.

    b. Run the program. When prompted to enter your age, type your age and press Enter. The message that appears should contain your age; however, it contains the number 0 instead. This is because the `age` variable is passed *by value* to the `getAge` function.

    c. Modify the program so that it passes the `age` variable *by reference* to the `getAge` function. Save and then run the program. When prompted to enter your age, type your age and press Enter. This time, the message contains your age.

TRY THIS

11. In this exercise, you modify the program from Lab 9-2 in Chapter 9. If necessary, create a new project named TryThis11 Project. Copy the instructions from the Lab9-2.cpp file into a source file named TryThis11.cpp. (The Lab9-2.cpp file is contained in either the Cpp6\Chap09\Lab9-2 Project folder or the Cpp6\Chap09 folder. Alternatively, you can enter the instructions from Figure 9-46 into the TryThis11.cpp file.) Change the filename in the first comment to TryThis11.cpp. Modify the program to use a void function named `displayPayment`. The `displayPayment` function should accept a monthly payment and then display the monthly payment on the screen. Save and then run the program. Test the program using 16000, 3000, .08, .03, and 4 as the car price, rebate, credit union rate, dealer rate, and term. The monthly payments should be $317.37 and $354.15. (The answers to TRY THIS Exercises are located at the end of the chapter.)

TRY THIS

12. In this exercise, you modify the program from TRY THIS Exercise 14 in Chapter 9. If necessary, create a new project named TryThis12 Project. If you completed Chapter 9's TRY THIS Exercise 14, copy the instructions from the TryThis14.cpp file into a source file named TryThis12.cpp. (The TryThis14.cpp file is contained in either the Cpp6\Chap09\TryThis14 Project folder or the Cpp6\Chap09 folder. Alternatively, you can enter the instructions from Figure 9-49 into the TryThis12.cpp file.) Change the filename in the first comment to TryThis12.cpp. Change the `getFahrenheit` and `calcCelsius` functions to void functions. Save and then run the program. Test the program using the following Fahrenheit temperatures: 32 and 212.

13. In this exercise, you modify the code from Computer Exercise 11. If necessary, create a new project named ModifyThis13 Project. Enter (or copy) the TryThis11.cpp instructions into a new source file named ModifyThis13.cpp. Change the filename in the first comment to ModifyThis13.cpp. Change the `getPayment` function to a void function named `calcPayment`. Save and then run the program. Test the program using 16000, 3000, .08, .03, and 4 as the car price, rebate, credit union rate, dealer rate, and term. The monthly payments should be $317.37 and $354.15.

MODIFY THIS

14. In this exercise, you modify the program from Lab 8-2 in Chapter 8. If necessary, create a new project named Introductory14 Project. Copy the instructions from the Lab8-2.cpp file into a source file named Introductory14.cpp. (The Lab8-2.cpp file is contained in either the Cpp6\Chap08\Lab8-2 Project folder or the Cpp6\Chap08 folder. Alternatively, you can enter the instructions from Figure 8-40 into the Introductory14.cpp file.) Change the filename in the first comment to Introductory14.cpp. Modify the program so that it uses a void function to display the multiplication table. Save and then run the program. Test the program using multiplicands of 2 and 4, followed by a sentinel value.

INTRODUCTORY

15. In this exercise, you modify the program from Lab 7-2 in Chapter 7. If necessary, create a new project named Introductory15 Project. Copy the instructions from the Lab7-2.cpp file into a source file named Introductory15.cpp. (The Lab7-2.cpp file is contained in either the Cpp6\Chap07\Lab7-2 Project folder or the Cpp6\Chap07 folder. Alternatively, you can enter the instructions from Figure 7-48 into the Introductory15.cpp file.) Change the filename in the first comment to Introductory15.cpp. Modify the program so that it uses a void function to determine the grade. Save and then run the program. Test the program using the following scores and sentinel value: 45, 40, 41, 96, 89, and -1. The total points and grade should be 311 and B, respectively.

INTRODUCTORY

16. In this exercise, you modify the program from Lab 6-2 in Chapter 6. If necessary, create a new project named Introductory16 Project. Copy the instructions from the Lab6-2.cpp file into a source file named Introductory16.cpp. (The Lab6-2.cpp file is contained in either the Cpp6\Chap06\Lab6-2 Project folder or the Cpp6\Chap06 folder. Alternatively, you can enter the instructions from Figure 6-32 into the Introductory16.cpp file.) Change the filename in the first comment to Introductory16.cpp. Modify the program so that it uses a void function to determine the commission. Save and then run the program. Test the program using the following sales amounts: 15000 and 250,000.

INTRODUCTORY

17. In this exercise, you modify the program from Lab 5-2 in Chapter 5. If necessary, create a new project named Intermediate17 Project. Copy the instructions from the Lab5-2.cpp file into a source file named Intermediate17.cpp. (The Lab5-2.cpp file is contained in either the Cpp6\Chap05\Lab5-2 Project folder or the Cpp6\Chap05 folder. Alternatively, you can enter the instructions from Figure 5-33 into the Intermediate17.cpp file.) Change the filename in the first comment

INTERMEDIATE

to Intermediate17.cpp. Modify the program so that it uses two void functions: one to determine the fat calories and the other to determine the fat percentage. Save and then run the program. Test the program appropriately.

INTERMEDIATE

18. In this exercise, you create a simple payroll program using a `main` function and four void functions.

    a. Figure 10-35 shows the IPO chart information for the payroll program. Complete the C++ instructions column for the `main` function, as well as for the three void functions named `calcFedTaxes`, `calcNetPay`, and `displayInfo`. The FWT (Federal Withholding Tax) rate is 20% of the weekly salary, and the FICA (Federal Insurance Contributions Act) rate is 8% of the weekly salary.

    b. If necessary, create a new project named Intermediate18 Project. Enter your C++ instructions into a source file named Intermediate18.cpp. Also enter appropriate comments and any additional instructions required by the compiler. Display the taxes and net pay with two decimal places.

    c. Save and then run the program. Test the program using 500 and 650 as the salary.

`main` function

IPO chart information	**C++ instructions**

Input

salary
FWT rate (.2)
FICA rate (.08)

Processing

none

Output

FWT
FICA
net pay

Algorithm

1. enter the salary
2. repeat while (the salary > 0)
   call the calcFedTaxes function to calculate the FWT and FICA, pass the function the salary, FWT rate, FICA rate, and the addresses of variables in which to store the FWT and FICA

   call the calcNetPay function to calculate the net pay, pass the function the salary, FWT, FICA, and the address of the variable in which to store the net pay

Figure 10-35 *(continues)*

(continued)

```
        call the displayInfo function to display
        the FWT, FICA, and net pay, pass the
        function the FWT, FICA, and net pay

        enter the salary
    end repeat
```

calcFedTaxes function

IPO chart information **C++ instructions**

Input

salary (formal parameter)
FWT rate (formal parameter)
FICA rate (formal parameter)
addresses of variables to store:
FWT (formal parameter)
FICA (formal parameter)

Processing

none

Output

FWT
FICA

Algorithm

1. calculate the FWT by multiplying the salary by the FWT rate
2. calculate the FICA by multiplying the salary by the FICA rate

calcNetPay function

IPO chart information **C++ instructions**

Input

salary (formal parameter)
FWT (formal parameter)
FICA (formal parameter)
address of a variable to store:
net pay (formal parameter)

Processing

none

Output

net pay

Algorithm

calculate the net pay by subtracting the FWT and FICA from the salary

Figure 10-35 *(continues)*

(continued)

displayInfo function

IPO chart information **C++ instructions**

Input

FWT (formal parameter)
FICA (formal parameter)
net pay (formal parameter)

Processing

none

Output

FWT
FICA
net pay

Algorithm

display the FWT, FICA, and net pay

Figure 10-35

INTERMEDIATE

19. The payroll manager at Gerston Blankets wants a program that calculates and displays the gross pay for each of the company's employees. It also should calculate and display the total gross pay. The payroll manager will enter the number of hours the employee worked and his or her pay rate. Employees working more than 40 hours should receive time and one-half for the hours over 40. Use a void function to determine an employee's gross pay. Use a value-returning function to accumulate the total gross pay. The program should display the total gross pay only after the payroll manager has finished entering the data for all the employees. Use a sentinel value to end the program.

    a. Create IPO charts for the problem, and then desk-check the algorithm using the following four sets of hours worked and pay rates: 35, $10.50; 43, $15; 32, $9.75; and 20, $6.45.

    b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 10-35. Then, code the algorithm into a program.

    c. Desk-check the program using the same data used to desk-check the algorithm.

    d. If necessary, create a new project named Intermediate19 Project. Enter your C++ instructions into a source file named Intermediate19.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

    e. Save and then run the program. Test the program using the same data used to desk-check the program.

INTERMEDIATE

20. The sales manager at Tompkins Company wants a program that calculates and displays each salesperson's commission, which is 10% of

his or her sales. It also should display the total commission. Use a value-returning function to get the amount sold by a salesperson. The amount sold may contain a decimal place. Also use three void functions: one to calculate the 10% commission, another to display the commission, and another to calculate the total commission. The program should display the commission and total commission in fixed-point notation with two decimal places. Display the total commission only after the sales manager has finished entering the sales amounts. Use a sentinel value to end the program.

a. Create IPO charts for the problem.

b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 10-35. Then, code the algorithm into a program.

c. If necessary, create a new project named Intermediate20 Project. Enter your C++ instructions into a source file named Intermediate20.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

d. Save and then run the program. Test the program using the following four sales amounts: 12000, 23000, 10000, and 25000. Then enter your sentinel value.

21. ADVANCED

In this exercise, you create a program that calculates the average of three test scores. The program should contain two value-returning functions (`main` and `calcAverage`) and two void functions (`getTestScores` and `displayAverage`). The `main` function should call the void `getTestScores` function to get three test scores. The test scores may contain a decimal place. The `main` function then should call the value-returning `calcAverage` function to calculate and return the average of the three test scores. When the `calcAverage` function has completed its task, the `main` function should call the void `displayAverage` function to display the average of the three test scores on the screen. Display the average with one decimal place. Use a sentinel value to end the program.

a. Create IPO charts for the problem, and then desk-check the algorithm using the following four sets of test scores, followed by your sentinel value: 56, 78, 90; 100, 85, 67; 74, 32, 98; and 25, 99, 84.

b. List the input, processing, and output items, as well as the algorithm, in a chart similar to the one shown earlier in Figure 10-35. Then, code the algorithm into a program.

c. Desk-check the program using the same data used to desk-check the algorithm.

d. If necessary, create a new project named Advanced21 Project. Enter your C++ instructions into a source file named Advanced21.cpp. Also enter appropriate comments and any additional instructions required by the compiler.

e. Save and then run the program. Test the program using the same data used to desk-check the program.

ADVANCED

22. In this exercise, you prevent a function from changing the value of a named constant passed to it.

    a. If necessary, create a new project named Advanced22 Project. Copy the instructions from the Lab10-2.cpp file into a source file named Advanced22.cpp. (Alternatively, you can enter the instructions from Figure 10-31 into the Advanced22.cpp file.) Change the filename in the first comment to Advanced22.cpp.

    b. The program passes the value of the `UNIT_CHG` named constant to the `calcBill` function, which stores the value it receives (.09) in the `chgPerUnit` variable. Because the function stores the value in a variable, the value can be changed by the function. First, you will verify that the `calcBill` function can change the value stored in the `chgPerUnit` variable. Insert a blank line above the `totChg = numUnits * chgPerUnit;` statement in the `calcBill` function. In the blank line, type `chgPerUnit = .25;`. Save and then run the program. Enter 3000 and 2000 as the current and previous readings. The program displays 1000 as the number of units used, which is correct. However, rather than displaying $90.00 as the total charge, the program displays $250.00.

    c. To prevent the `calcBill` function from changing the charge per unit value, you must use the `const` keyword to indicate that the value being passed is a constant. Make the appropriate modification to the `calcBill` function's prototype and its header. Save and then run the program. The compiler displays an error message indicating that you cannot assign a value to the `chgPerUnit` variable, which is defined using the `const` keyword.

    d. Delete the `chgPerUnit = .25;` statement from the `calcBill` function. Save and then run the program. Enter 3000 and 2000 as the current and previous readings. The program correctly displays 1000 and $90.00 as the units used and total charge, respectively.

ADVANCED

23. In this exercise, you modify the electric bill program from Lab10-2.

    a. If necessary, create a new project named Advanced23 Project. Copy the instructions from the Lab10-2.cpp file into a source file named Advanced23.cpp. (Alternatively, you can enter the instructions from Figure 10-31 into the Advanced23.cpp file.) Change the filename in the first comment to Advanced23.cpp.

    b. Modify the program so that it allows the user to display the electric bill for more than one customer without having to run the program again. Use a sentinel value to end the program. The sentinel value should be entered as the current meter reading in the `getInput` function. When the user enters the sentinel value, the `getInput` function should not prompt the user to enter the previous reading.

    c. Change the `getInput` function to a value-returning function. If the user does not enter a sentinel value as the current reading, the `getInput` function should prompt the user to enter the previous

reading. It then should return a value that indicates whether the current and previous readings are valid or invalid. To be valid, the current reading must be greater than or equal to the previous reading. The `main` function should not call the `calcBill` or `displayBill` functions when the readings are not valid. Instead, it should display an error message.

d. Save and then run the program. Enter 3000 and 2000 as the current and previous readings. The program should display 1000 and $90.00 as the units used and total charge, respectively, and then prompt you to enter the current reading. Now enter 3000 and 5000 as the current and previous readings. The program should display an error message and then prompt you to enter the current reading. Enter 53512 and 51875 as the current and previous readings. The program should display 1637 and $147.33 as the units used and total charge, respectively. Enter your sentinel value to end the program.

24. Follow the instructions for starting C++ and opening the SwatTheBugs24.cpp file. The file is contained in either the Cpp6\Chap10\SwatTheBugs24 Project folder or the Cpp6\Chap10 folder. Run the program. Enter 1000 and .1 as the sales and bonus rate. Debug the program.

SWAT THE BUGS

25. Follow the instructions for starting C++ and opening the SwatTheBugs25.cpp file. The file is contained in either the Cpp6\Chap10\SwatTheBugs25 Project folder or the Cpp6\Chap10 folder. Debug the program.

SWAT THE BUGS

Answers to TRY THIS Exercises



Pencil and Paper

1.
```
void divideByTwo(int wholeNumber)
{
     cout << wholeNumber / 2.0;
}    //end of divideByTwo function
```

2. `void divideByTwo(int);` or `void divideByTwo(int wholeNumber);`

3. `divideByTwo(total);`



Computer

10. To modify the program, change the function prototype to `void getAge(int &years);` and change the function header to `void getAge(int &years)`.

11. See Figure 10-36. The modifications to the program are shaded in the figure.

```
1  //TryThis11.cpp - displays two monthly car payments
2  //Created/revised by <your name> on <current date>
3
4  #include <iostream>
5  #include <cmath>
6  #include <iomanip>
7  using namespace std;
8
9  //function prototypes
10 double getPayment(int, double, int);
11 void displayPayment(double);
12
13 int main()
14 {
15     //declare variables
16     int carPrice         = 0;
17     int rebate           = 0;
18     double creditRate    = 0.0;
19     double dealerRate    = 0.0;
20     int term             = 0;
21     double creditPayment = 0.0;
22     double dealerPayment = 0.0;
23
24     //get input items
25     cout << "Car price (after any trade-in): ";
26     cin >> carPrice;
27     cout << "Rebate: ";
28     cin >> rebate;
29     cout << "Credit union rate: ";
30     cin >> creditRate;
31     cout << "Dealer rate: ";
32     cin >> dealerRate;
33     cout << "Term in years: ";
34     cin >> term;
35
36     //call function to calculate payments
37     creditPayment = getPayment(carPrice - rebate,
38              creditRate / 12, term * 12);
39     dealerPayment = getPayment(carPrice,
40              dealerRate / 12, term * 12);
41
42     //display payments
43     cout << fixed << setprecision(2) << endl;
44     cout << "Credit union payment: $";
45     displayPayment(creditPayment);
46     cout << endl;
47     cout << "Dealer payment: $";
48     displayPayment(dealerPayment);
49     cout << endl;
```

you also can use `void displayPayment(double mthlyPay);`

Figure 10-36 *(continues)*

(continued)

```
50
51      system("pause");
52      return 0;
53  }   //end of main function
54
55  //*****function definitions*****
56  double getPayment(int prin,
57                    double monthRate,
58                    int months)
59  {
60      //calculates and returns a monthly payment
61      double monthPay = 0.0;
62      monthPay = prin * monthRate /
63                (1 - pow(monthRate + 1, -months));
64      return monthPay;
65  }   //end of getPayment function
66
67  void displayPayment(double mthlyPay)
68  {
69      cout << mthlyPay;
70  }   //end of displayPayment function
```

your C++ development tool may not require this statement

Figure 10-36

12. See Figure 10-37. The modifications to the program are shaded in the figure. (Several lines of code also were removed from the two void functions.)

```
 1  //TryThis12.cpp - converts Fahrenheit to Celsius
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <iomanip>
 6  using namespace std;
 7
 8  //function prototypes
 9  void getFahrenheit(int &tempF);
10  void calcCelsius(int tempF, double &tempC);
11
12  int main()
13  {
14      int fahrenheit = 0;
15      double celsius = 0.0;
16
17      //get input item
18      getFahrenheit(fahrenheit);
19
20      //calculate Celsius
21      calcCelsius(fahrenheit, celsius);
22
23      //display output item
```

Figure 10-37 *(continues)*

(continued)

```
24      cout << fixed << setprecision(0);
25      cout << "Celsius: " << celsius << endl;
26
27      system("pause");
28      return 0;
29  }   //end of main function
30
31  //*****function definitions*****
32  void getFahrenheit(int &tempF)
33  {
34      cout << "Enter Fahrenheit temperature: ";
35      cin >> tempF;
36  }   //end of getFahrenheit function
37
38  void calcCelsius(int tempF, double &tempC)
39  {
40      tempC = 5.0 / 9.0 * (tempF - 32.0);
41  }   //end of calcCelsius function
```

your C++ development tool may not require this statement

Figure 10-37

CHAPTER 11

One-Dimensional Arrays

After studying Chapter 11, you should be able to:

- Declare and initialize a one-dimensional array
- Enter data into a one-dimensional array
- Display the contents of a one-dimensional array
- Pass a one-dimensional array to a function
- Calculate the total and average of the values in a one-dimensional array
- Search a one-dimensional array
- Access an individual element in a one-dimensional array
- Find the highest value in a one-dimensional array
- Explain the bubble sort algorithm
- Use parallel one-dimensional arrays

Arrays

All of the variables you have used so far have been simple variables. A **simple variable**, also called a **scalar variable**, is one that is unrelated to any other variable in memory. At times, however, you will encounter situations in which some of the variables in a program *are* related to each other. In those cases, it is easier and more efficient to treat the related variables as a group. You already are familiar with the concept of grouping. The clothes in your closet probably are separated into groups, such as coats, sweaters, shirts, and so on. Grouping your clothes in this manner allows you to easily locate your favorite sweater, because you just need to look through the sweater group rather than through the entire closet. You also probably have your CD (compact disc) collection grouped by either music type or artist. If your collection is grouped by artist, it will take only a few seconds to find all of your Beatles CDs and, depending on the number of Beatles CDs you own, only a short time after that to locate a particular CD. When you group together related variables that have the same data type, the group is referred to as an array of variables or, more simply, an **array**. You might use an array of 50 variables to store the population of each U.S. state. Or, you might use an array of four variables to store the sales made in each of your company's four sales regions. Storing data in an array increases the efficiency of a program, because data can be both stored in and retrieved from the computer's internal memory much faster than it can be written to and read from a file on a disk. In addition, after the data is entered into an array, which typically is done at the beginning of a program, the program can use the data as many times as necessary without having to enter the data again. Your company's sales program, for example, can use the sales amounts stored in an array to calculate the total company sales and the percentage that each region contributed to the total sales. It also can use the sales amounts in the array either to calculate the average sales amount or to simply display the sales made in a specific region. As you will learn in this chapter, the variables in an array can be used just like any other variables. You can assign values to them, use them in calculations, display their contents, and so on.



It takes longer for the computer to access the information stored in a disk file, because the computer must wait for the disk drive to first locate the needed information and then read the information into internal memory.

The most commonly used arrays in business applications are one-dimensional and two-dimensional. You will learn about one-dimensional arrays in this chapter. Two-dimensional arrays are covered in Chapter 12. Arrays having more than two dimensions are used mostly in scientific and engineering programs and are beyond the scope of this book. As is true of functions, which you learned about in Chapters 9 and 10, arrays are one of the more challenging topics for beginning programmers. Therefore, it is important for you to read and study each section in this chapter thoroughly before moving on to the next section. For example, be sure you understand the concept of one-dimensional arrays before you continue to the sections pertaining to the bubble sort and parallel arrays. If you still feel overwhelmed by the end of the chapter, try reading the chapter again, paying particular attention to the examples and programs shown in the figures.

One-Dimensional Arrays

The variables in an array are stored in consecutive locations in the computer's internal memory. Each variable in an array has the same name and data

type. You distinguish one variable in a **one-dimensional array** from another variable in the same array using a unique number. The unique number, which is always an integer, is called a subscript. The **subscript** indicates the variable's position in the array and is assigned by the computer when the array is created in internal memory. The first variable in a one-dimensional array is assigned a subscript of 0, the second a subscript of 1, and so on. You refer to each variable in an array by the array's name and the variable's subscript, which is specified in a set of square brackets immediately following the array name. To refer to the first variable in a one-dimensional array named `sales`, you use `sales[0]`—read "`sales` sub zero." Similarly, to refer to the second variable in the `sales` array, you use `sales[1]`. If the `sales` array contains four variables, you refer to the fourth (and last) variable using `sales[3]`. Notice that the last subscript in an array is always one number less than the total number of variables in the array; this is because array subscripts start at 0. Figure 11-1 illustrates the variables contained in the one-dimensional `sales` array using the storage bin analogy from Chapter 3. The `age`, `rate`, and `unitCharge` variables included in the figure are scalar variables.



A subscript also is called in index.

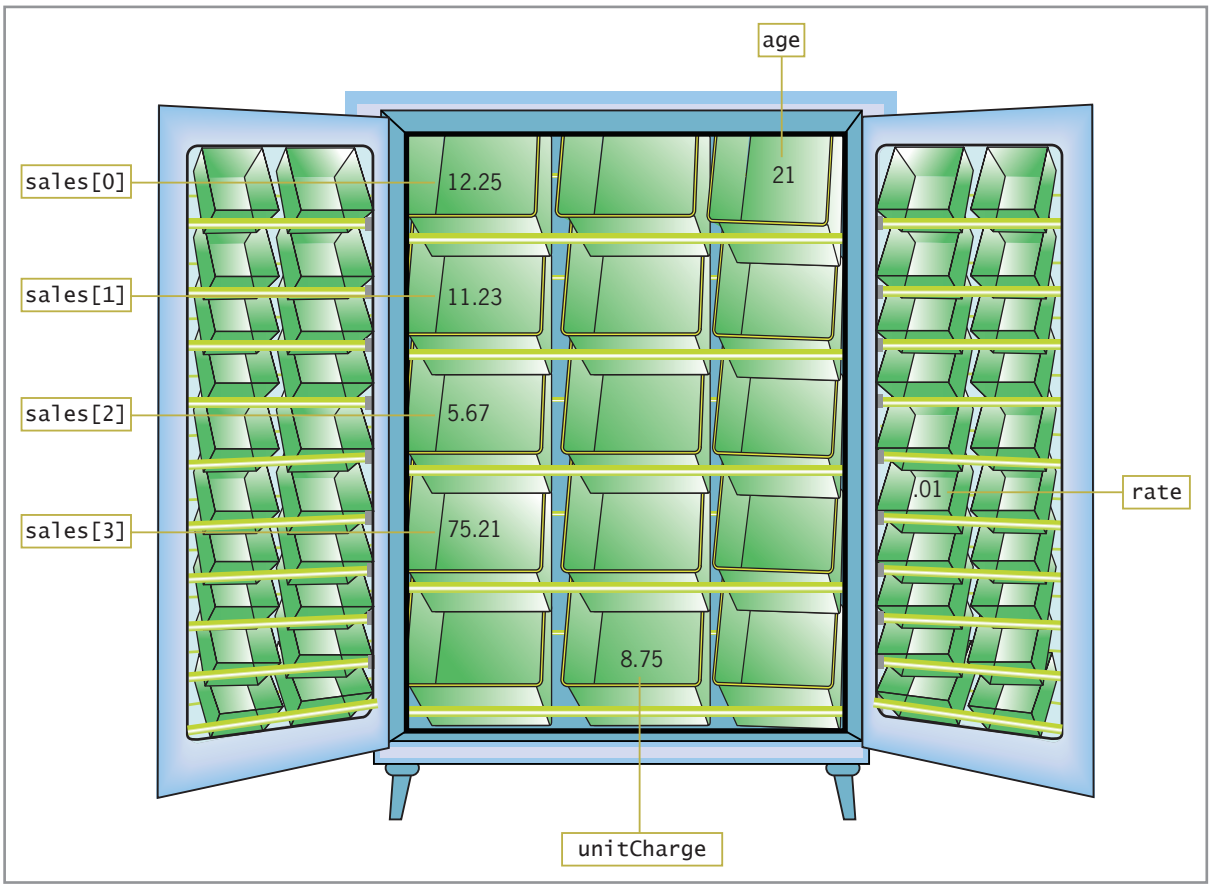


Figure 11-1 Illustration of the naming convention for the one-dimensional `sales` array

Figure 11-2 shows the problem specification and IPO chart information for the XYZ Company's sales program, which uses an array to store the sales made in each of the company's four regions. (The flowchart for this program is contained in the Ch11Flowcharts.pdf file, which is located in the Cpp6\Chap11 folder.) The program allows the user to enter the sales amounts and

then displays the amounts on the computer screen. Before you can code the sales program, you need to learn how to declare and initialize a one-dimensional array. You also need to learn how to store data in an array and display data from an array.

Problem specification
The XYZ Company wants a program that allows the user to enter the sales made in each of its four sales regions. The program then should display the amounts on the computer screen.

Input	Processing	Output
sales (made in each of 4 regions)	Processing items: array (4 elements) subscript counter (0 to 3) Algorithm: 1. repeat for (each of the 4 array elements) enter the sales into the current array element end repeat 2. repeat for (each of the 4 array elements) display the sales stored in the current array element end repeat	sales (made in each of 4 regions)

Figure 11-2 Problem specification and IPO chart for the XYZ Company's sales program

Declaring and Initializing a One-Dimensional Array

Before you can use an array in a program, you first must declare (create) it. It also is a good programming practice to initialize the array variables to ensure they will not contain garbage when the program is run. As you learned in Chapter 3, the garbage found in uninitialized variables is the remains of what was last stored at the memory location that the variable now occupies. Figure 11-3 shows the syntax for declaring a one-dimensional array in C++ and initializing its variables. In the syntax, *dataType* is the type of data that the array variables, referred to as **elements**, will store. Recall that each of the elements (variables) in an array has the same data type. *ArrayName* in the syntax is the name of the array. You use the same rules for naming an array as you do for naming a variable. *NumberOfElements* is an integer that specifies the size of the array. In other words, it specifies the number of elements you want in the array. To declare an array that contains 10 elements, you enter the number 10 as the *numberOfElements*. Notice that you enclose the *numberOfElements* in square brackets ([]). You can initialize the array elements at the same time you declare the array simply by entering one or more values, separated by commas, in the *initialValues* section of the syntax. You enclose the *initialValues* section in braces ({}). Assigning initial values to an array is often referred to as **populating the array**. The values used to populate an array should have the same data type as the array variables. If the data types are not the same, the computer uses implicit type conversion to either promote or demote the values to fit the array variables. However, recall from Chapter 3 that the implicit demotion of values can adversely affect a program's output. Therefore, you always should be sure to populate an array using values that have the appropriate data type. Also included

in Figure 11-3 are examples of declaring and initializing one-dimensional arrays. The declaration statement in Example 1 creates a three-element `char` array named `letters`. It initializes the `letters[0]` element to A, the `letters[1]` element to B, and the `letters[2]` element to C. Example 2 shows two statements you can use to declare a four-element `double` array, initializing each element to the `double` number 0.0. The statement `double sales[4] = {0.0, 0.0, 0.0, 0.0};` provides an initial value for each of the four array elements, whereas the statement `double sales[4] = {0.0};` provides only one value. When the array declaration statement does not provide an initial value for each of the elements in a numeric array, most C++ compilers initialize the uninitialized array elements to either 0.0 or 0 (depending on the data type of the array). However, this is done only when you provide at least one value in the *initialValues* section. If you omit the *initialValues* section from the declaration statement—for example, if you use the statement `double sales[4];` to declare the array—the compiler does not automatically initialize the elements, so the array elements may contain garbage. Example 3 shows two statements you can use to declare a six-element `int` array named `numbers`. The statement `int numbers[6] = {12, 0, 0, 0, 0, 0};` initializes the first array element to the integer 12 and initializes the remaining elements to the integer 0. The same result can be accomplished using the `int numbers[6] = {12};` statement shown in the example.



Most C++ compilers initialize `char` array elements to a space, `string` array elements to the empty string, and `bool` array elements to the keyword `false`.



The = **{***initialValues***}** portion of the syntax in Figure 11-3 is optional. Typically, optional items are enclosed in square brackets when shown in the syntax. In this case, the square brackets were omitted so as not to confuse them with the square brackets that are required by the syntax.

HOW TO Declare and Initialize a One-Dimensional Array

Syntax

*dataType arrayName***[***numberOfElements***] = {***initialValues***};**

Example 1

```
char letters[3] = {'A', 'B', 'C'};
```

declares and initializes a three-element `char` array named `letters`

Example 2

```
double sales[4] = {0.0, 0.0, 0.0, 0.0};
                or
double sales[4] = {0.0};
```

declares and initializes a four-element `double` array named `sales`; each element is initialized to 0.0

Example 3

```
int numbers[6] = {12, 0, 0, 0, 0, 0};
               or
int numbers[6] = {12};
```

declares and initializes a six-element `int` array named `numbers`; the first element is initialized to 12, whereas the others are initialized to 0

Figure 11-3 How to declare and initialize a one-dimensional array

Keep in mind that if you inadvertently provide more values in the *initialValues* section than the number of array elements, most C++ compilers will display the error message "too many initializers" when you

attempt to compile the program. However, not all C++ compilers display a message when this error occurs. Rather, some compilers store the extra values in memory locations adjacent to, but not reserved for, the array. Therefore, you always should be careful to provide no more than the appropriate number of *initialValues*.

Entering Data into a One-Dimensional Array

As you can with a scalar variable, you can use either an assignment statement or the extraction operator to enter data into an array element. Figure 11-4 shows the syntax of such an assignment statement. In the syntax, *arrayName*[*subscript*] is the name and subscript of the array variable to which you want the *expression* (data) assigned. The *expression* can include any combination of constants, variables, and operators. The data type of the *expression* must match the data type of the array element to which the *expression* is assigned; otherwise, an implicit type conversion will occur, which could result in incorrect output. Also included in Figure 11-4 are examples of assignment statements that assign data to the elements in various arrays. The arrays were declared earlier in Figure 11-3. The assignment statement in Example 1 assigns the letter Y to the second element in the `letters` array, replacing the letter B that was stored in the element when the array was declared. The code in Example 2 assigns the `double` number 0.0 to each of the four elements in the `sales` array and provides another means of initializing the array. The code in Example 3 assigns the squares of the numbers from 1 through 6 to the six-element `numbers` array, replacing the array's initial values. The square of the number 1 is assigned to the `numbers[0]` element. The square of the number 2 is assigned to the `numbers[1]` element, and so on. Notice that the `x` variable keeps track of the six numbers to be squared. Also notice that, in order to assign the square of each number to its appropriate element in the `numbers` array, the code must subtract the number 1 from the value stored in the `x` variable. This is because the `x` variable's values go from 1 through 6, whereas their corresponding array subscripts go from 0 through 5. Example 4's code updates the contents of each element in the six-element `numbers` array. It does this by adding the value contained in the `increase` variable to the value contained in the current array element and then assigning the sum to the current array element. The loops in Examples 2 through 4 provide a convenient way to access each element in a one-dimensional array.

HOW TO Use an Assignment Statement to Assign Data to a One-Dimensional Array

Syntax

arrayName[*subscript*] = *expression*;

Example 1

```
letters[1] = 'Y';
```

assigns the letter Y to the second element in the `letters` array

Figure 11-4 How to use an assignment statement to assign data to a one-dimensional array *(continues)*

(continued)

Example 2

```
int subscript = 0;
while (subscript < 4)
{
      sales[subscript] = 0.0;
      subscript += 1;
}     //end while
```

assigns the `double` number 0.0 to each of the four elements in the `sales` array; provides another means of initializing the array

Example 3

```
for (int x = 1; x <= 6; x += 1)
      numbers[x - 1] = pow(x, 2);
//end for
```

assigns the squares of the numbers from 1 through 6 to the six-element `numbers` array

Example 4

```
int increase = 0;
cout << "Enter increase amount: ";
cin >> increase;
for (int x = 0; x < 6; x += 1)
      numbers[x] += increase;
//end for
```

assigns, to each element in the six-element `numbers` array, the sum of the element's current value plus the value stored in the `increase` variable

Figure 11-4 How to use an assignment statement to assign data to a one-dimensional array

As already mentioned, you also can use the extraction operator to store data in an array element; this is shown in the syntax and examples in Figure 11-5. (The arrays in Figure 11-5 were declared earlier in Figure 11-3.) The `cin` statement in Example 1 stores the user's entry in the first element in the `letters` array, replacing the element's existing data. Example 2 contains a loop that repeats its instructions four times: once for each element in the `sales` array. The loop instructions prompt the user to enter a sales amount and then store the user's response in the current element. Example 3 contains a loop that repeats its instructions for each of the six elements in the `numbers` array. The loop instructions prompt the user to enter an integer and then store the user's response in the current element.

HOW TO Use the Extraction Operator to Store Data in a One-Dimensional Array

Syntax

cin >> arrayName[*subscript*];

Example 1

```
cin >> letters[0];
```

stores the user's entry in the first element in the `letters` array

Example 2

```
for (int sub = 0; sub < 4; sub += 1)
{
      cout << "Enter the sales for Region ";
      cout << sub + 1 << ": ";
      cin >> sales[sub];
}     //end for
```

stores the user's entries in the four-element `sales` array

Example 3

```
int x = 0;
while (x < 6)
{
      cout << "Enter an integer: ";
      cin >> numbers[x];
      x += 1;
}     //end while
```

stores the user's entries in the six-element `numbers` array

Figure 11-5 How to use the extraction operator to store data in a one-dimensional array

Displaying the Contents of a One-Dimensional Array

To display the contents of an array, you need to access each of its elements. You do this using a loop along with a counter variable that keeps track of each subscript in the array. Figure 11-6 shows examples of loops you can use to display the contents of the arrays declared earlier in Figure 11-3. Example 1 uses the `while` statement to display the contents of the `letters` array, which contains three elements. Example 2 uses the `for` statement to display the contents of the four-element `sales` array, and Example 3 uses the `do while` statement to display the contents of the six-element `numbers` array.

HOW TO Display the Contents of a One-Dimensional Array

Example 1

```
int x = 0;
while (x < 3)
}
      cout << letters[x] << endl;
      x += 1;
}     //end while
```

displays the contents of the three-element `letters` array

Example 2

```
for (int sub = 0; sub < 4; sub += 1)
{
      cout << "Sales for Region " << sub + 1 << ": $";
      cout << sales[sub] << endl;
}     //end for
```

displays the contents of the four-element `sales` array

Example 3

```
int x = 0;
do    //begin loop
{
      cout << numbers[x] << endl;
      x += 1;
} while (x < 6);
```

displays the contents of the six-element `numbers` array



Notice that the valid subscripts for the `sales` array in Example 2 are 0 through 3, whereas the valid region numbers are 1 through 4.

Figure 11-6 How to display the contents of a one-dimensional array

Now that you know how to declare and initialize a one-dimensional array, as well as how to store data in the array and display data from the array, you can code the XYZ Company's sales program.

Coding the XYZ Company's Sales Program

Earlier, in Figure 11-2, you viewed the problem specification and IPO chart for the XYZ Company's sales program. Figure 11-7 shows the IPO chart information along with the corresponding C++ instructions. Figure 11-8 shows the code for the entire program, and Figure 11-9 shows a sample run of the program.

IPO chart information	C++ instructions
Input	
sales (made in each of 4 regions)	the sales will be entered into the array
Processing	
array (4 elements)	`double sales[4] = {0.0};`
subscript counter (0 to 3)	declared and initialized in the for clause
Output	
sales (made in each of 4 regions)	displayed from the array by the for loop
Algorithm	

1. repeat for (each of the 4 array elements)
enter the sales into the current array element
 	
end repeat
	`for (int sub = 0, sub < 4; sub += 1)`
	`{`
	`  cout << "Enter the sales`
	`  for Region ";`
	`  cout << sub + 1 << ": ";`
	`  cin >> sales[sub];`
	`}  //end for`

2. repeat for (each of the 4 array elements)
display the sales stored in the current array element
end repeat
	`for (int sub = 0, sub < 4; sub += 1)`
	`{`
	`  cout << "Sales for Region "`
	`  << sub + 1 << ": $";`
	`  cout << sales[sub] << endl;`
	`}  //end for`

Figure 11-7 IPO chart information and C++ instructions for the XYZ Company's sales program

```
 1  //XYZ Company.cpp - displays the contents of an array
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <iomanip>
 6  using namespace std;
 7
 8  int main()
 9  {
10      //declare array
11      double sales[4] = {0.0};
12
13      //fill array with data
14      for (int sub = 0; sub < 4; sub += 1)
15      {
16            cout << "Enter the sales for Region ";
17            cout << sub + 1 << ": ";
18            cin >> sales[sub];
19      }     //end for
20
```

array declaration

stores data in the array

Figure 11-8 XYZ Company's sales program *(continues)*

(continued)

```
21      //display the contents of the array
22      cout << fixed << setprecision(2) << endl << endl;
23      for (int sub = 0; sub < 4; sub += 1)
24      {
25            cout << "Sales for Region " << sub + 1 << ": $";
26            cout << sales[sub] << endl;
27      }     //end for
28
29      system("pause");
30      return 0;
31  }   //end of main function
```

displays the contents of the array

your C++ development tool may not require this statement

Figure 11-8 XYZ Company's sales program

```
XYZ Company Sales Program
Enter the sales for Region 1: 123.67
Enter the sales for Region 2: 350.89
Enter the sales for Region 3: 567
Enter the sales for Region 4: 224.98


Sales for Region 1: $123.67
Sales for Region 2: $350.89
Sales for Region 3: $567.00
Sales for Region 4: $224.98
Press any key to continue . . .
```

Figure 11-9 Sample run of the XYZ Company's sales program

Desk-checking the code in Figure 11-8 will help you understand how arrays operate in a program. You will desk-check the code using the following four sales amounts: 123.67, 350.89, 567, and 224.98. First, the declaration statement on Line 11 declares and initializes a four-element `double` array named `sales`. Figure 11-10 shows the desk-check table after the declaration statement is processed.

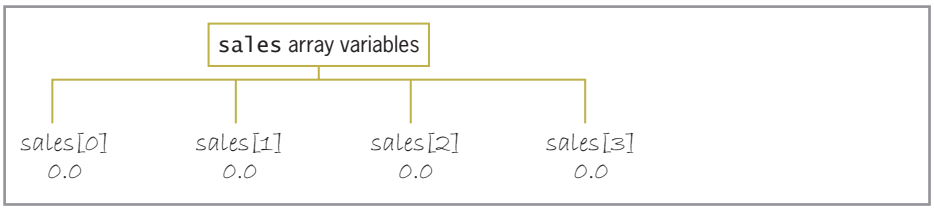


Figure 11-10 Desk-check table after the array declaration statement is processed

The `for` clause on Line 14 is processed next. The clause's *initialization* argument declares an `int` variable named `sub` and initializes it to the number 0. The `sub` variable is a counter variable that will keep track of the four array subscripts: 0, 1, 2, and 3. As you learned in Chapter 7, a variable declared in a `for` clause is local to the `for` loop and can be used only by the statements within the loop. In this case, the `sub` variable is local to the `for` loop on

Lines 14 through 19. The `sub` variable will remain in memory until the `for` loop ends. Figure 11-11 shows the desk-check table after the *initialization* argument on Line 14 has been processed.

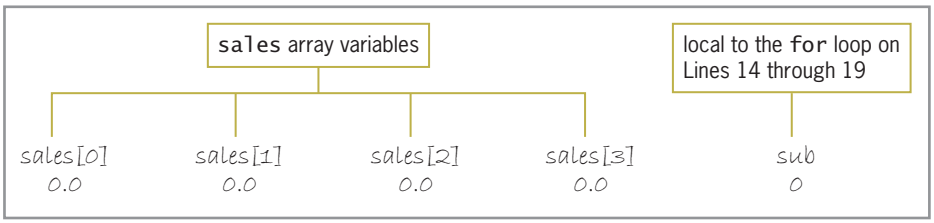


Figure 11-11 Desk-check table after the *initialization* argument on Line 14 is processed

The `for` clause's *condition* argument tells the computer to check whether the value stored in the `sub` variable is less than 4. It is, so the statements in the body of the `for` loop are processed. First, the `cout` statements on Lines 16 and 17 prompt the user to enter the sales for the current region—in this case, Region 1. Notice that the current region is determined by adding the number 1 to the value stored in the `sub` variable (0). This is because, unlike the array subscripts, the region numbers go from 1 through 4 rather than from 0 through 3. The region number always will be one number more than the subscript of its corresponding element in the array. In other words, Region 1's sales will be stored in the element whose subscript is 0. Likewise, Region 2's sales will be stored in the element whose subscript is 1, and so on. The `cin` statement on Line 18 gets Region 1's sales from the user and then stores the amount in the first array element (`sales[0]`). Figure 11-12 shows the sales for Region 1 (123.67) entered in the array.



As you learned in Chapter 7, the *condition* argument in a `for` clause is a looping condition, because it specifies the requirement for processing the loop instructions.

sales[0]	sales[1]	sales[2]	sales[3]	sub
~~0.0~~ 123.67	0.0	0.0	0.0	0

Figure 11-12 Desk-check table after Region 1's sales are entered in the array

The `for` clause's *update* argument tells the computer to add the number 1 to the contents of the `sub` variable; the result is 1. The computer then checks whether the `sub` variable's value is less than 4. It is, so the statements in the body of the `for` loop are processed again. The `cout` statements prompt the user to enter the sales for Region 2, and the `cin` statement stores the user's response in the second array element (`sales[1]`). Figure 11-13 shows the sales for Region 2 (350.89) entered in the array.

sales[0]	sales[1]	sales[2]	sales[3]	sub
~~0.0~~ 123.67	~~0.0~~ 350.89	0.0	0.0	~~0~~ 1

Figure 11-13 Desk-check table after Region 2's sales are entered in the array

Next, the computer updates the `sub` variable by adding the number 1 to it; the result is 2. The computer then checks whether the `sub` variable's value is less than 4. It is, so the statements in the body of the `for` loop are processed

again. The `cout` statements prompt the user to enter the sales for Region 3, and the `cin` statement stores the user's response in the third array element (`sales[2]`). Figure 11-14 shows the sales for Region 3 (567.0) entered in the array.

sales[0]	sales[1]	sales[2]	sales[3]	sub
~~0.0~~	~~0.0~~	~~0.0~~	0.0	~~0~~
123.67	350.89	567.0		~~1~~
				2

Figure 11-14 Desk-check table after Region 3's sales are entered in the array

Next, the computer updates the `sub` variable by adding the number 1 to it; the result is 3. The computer then checks whether the `sub` variable's value is less than 4. It is, so the statements in the body of the `for` loop are processed again. The `cout` statements prompt the user to enter the sales for Region 4, and the `cin` statement stores the user's response in the fourth array element (`sales[3]`). Figure 11-15 shows the sales for Region 4 (224.98) entered in the array.

sales[0]	sales[1]	sales[2]	sales[3]	sub
~~0.0~~	~~0.0~~	~~0.0~~	~~0.0~~	~~0~~
123.67	350.89	567.0	224.98	~~1~~
				~~2~~
				3

Figure 11-15 Desk-check table after Region 4's sales are entered in the array

Once again, the computer updates the `sub` variable by adding the number 1 to it; this time, the result is 4. The computer then checks whether the `sub` variable's value is less than 4. It's not, so the `for` loop on Lines 14 through 19 ends and the computer removes the loop's local `sub` variable from internal memory. Figure 11-16 shows the current status of the desk-check table.

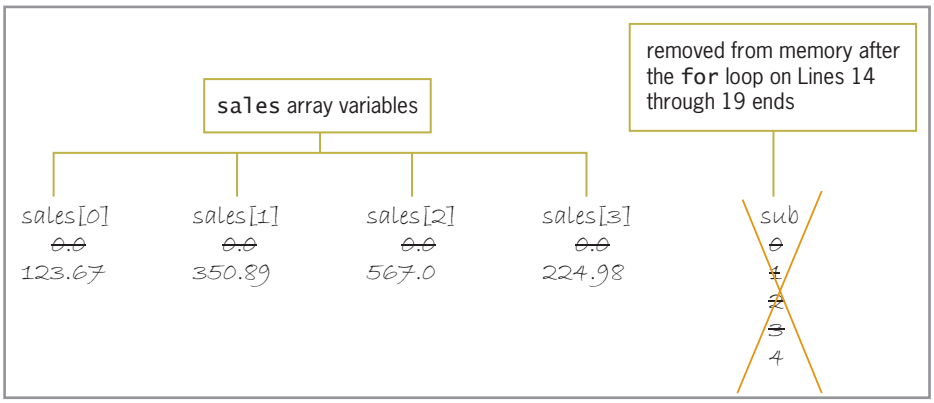


Figure 11-16 Desk-check table after the `for` loop on Lines 14 through 19 ends

The `cout` statement on Line 22 is processed next. The statement tells the computer to display real numbers in fixed-point notation with two decimal places; it also displays two blank lines. Next, the computer processes the `for` clause on Line 23. The clause's *initialization* argument declares and

initializes an `int` variable named `sub`. Although the variable's name is the same as the one in the first `for` clause, which appears on Line 14, it's not the same variable. This `sub` variable is local to the `for` loop on Lines 23 through 27. The `sub` variable created by the `for` clause on Line 14 was local to the `for` loop on Lines 14 through 19 and was removed from memory when that loop ended. Figure 11-17 shows the desk-check table after the *initialization* argument on Line 23 is processed.

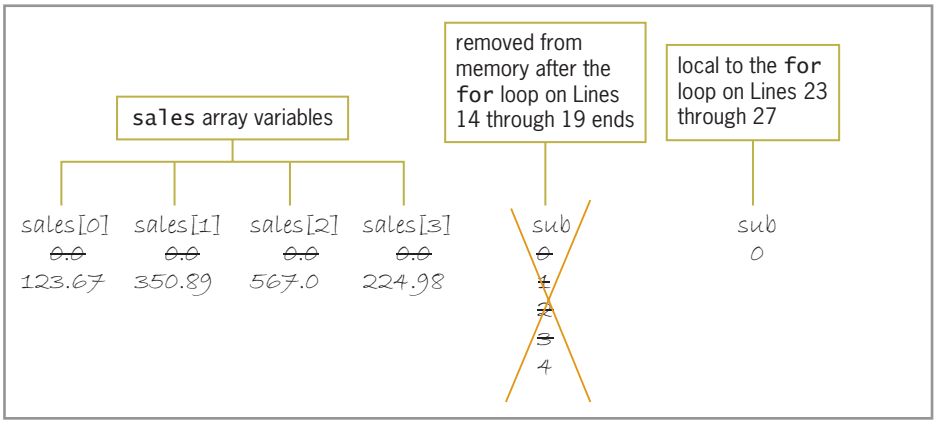


Figure 11-17 Desk-check table after the *initialization* argument on Line 23 is processed

The *condition* argument in the `for` clause on Line 23 tells the computer to check whether the value in the `sub` variable is less than 4. It is, so the statements in the body of the `for` loop are processed. Those statements display Region 1's sales, which are located in the `sales[0]` element in the array, on the computer screen. Next, the `for` clause's *update* argument adds the number 1 to the contents of the `sub` variable; the result is 1. The computer then checks whether the value in the `sub` variable is less than 4. It is, so the statements in the body of the `for` loop display Region 2's sales. Here again, the computer updates the `sub` variable by adding the number 1 to it; the result is 2. The computer then checks whether the value in the `sub` variable is less than 4. It is, so the statements in the body of the `for` loop display Region 3's sales. The computer again updates the `sub` variable by adding the number 1 to it; the result is 3. The computer then checks whether the value in the `sub` variable is less than 4. It is, so the statements in the body of the `for` loop display Region 4's sales. Once again, the computer updates the `sub` variable by adding the number 1 to it; this time, the result is 4. The computer then checks whether the value in the `sub` variable is less than 4. It's not, so the `for` loop ends and the computer removes the loop's local `sub` variable from internal memory. Figure 11-18 shows the desk-check table after the `for` loop on Lines 23 through 27 ends.

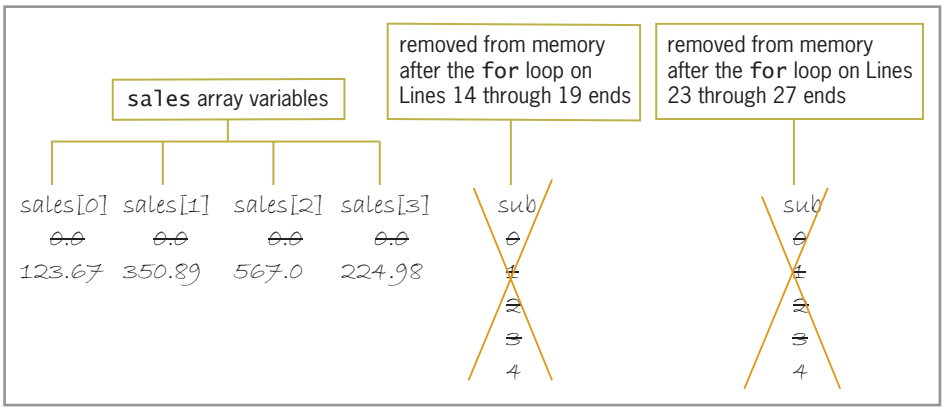


Figure 11-18 Desk-check table after the `for` loop on Lines 23 through 27 ends

Finally, the computer processes the `system("pause");` and `return 0;` statements on Lines 29 and 30. When the program ends, the computer removes the `sales` array from its internal memory.

Passing a One-Dimensional Array to a Function

Figure 11-19 shows a modified version of the XYZ Company's sales program. In the modified version, the `main` function passes the `sales` array to a program-defined void function named `displayArray`. The changes made to the original code (shown earlier in Figure 11-8) are shaded in Figure 11-19. Study closely the `displayArray` function prototype, function call, and function header. The function call, which appears on Line 26, passes two items of information to the `displayArray` function: the `sales` array and the number of elements in the array. You pass an array simply by including the array's name—in this case, `sales`—as the actual argument. As you know, variables can be passed to a function either *by value* or *by reference*. Unless specified otherwise, scalar variables in C++ are passed *by value*. To pass a scalar variable *by reference*, recall that you need to include the address-of (&) operator before the formal parameter's name in the receiving function's header. You also need to include the address-of operator in the receiving function's prototype. Unlike scalar variables, arrays in C++ are passed automatically *by reference* rather than *by value*; this is because it is more efficient to pass arrays in that manner. Since many arrays are large, passing an array *by value* would consume a great deal of the computer's memory and time, because the computer would need to duplicate the array in the receiving function's formal parameter. Passing an array *by reference* allows the computer to pass the address of only the first array element. Because array elements are stored in contiguous locations in memory, the computer can use the address to locate the remaining elements in the array. Given that arrays are passed automatically *by reference*, you do not include the address-of (&) operator before the formal parameter's name in the function header, as you do when passing scalar variables *by reference*. You also do not include the address-of operator in the function prototype. Instead, you indicate that you are passing an array to a function by simply entering the formal parameter's data type and name, followed by an empty set of square brackets, in the receiving function's header and in its prototype, as shown in Figure 11-19. (Recall that the formal parameter's name is optional in the prototype. Therefore, you also could write the function prototype in Figure 11-19 as `double []`.)



You learned about passing *by value* and *by reference* in Chapters 9 and 10.

```
1  //Modified XYZ Company.cpp - displays the contents
2  //of an array
3  //Created/revised by <your name> on <current date>
4
5  #include <iostream>
6  #include <iomanip>
7  using namespace std;
8
9  //function prototype
10 void displayArray(double dollars[], int numElements);
11
12 int main()
13 {
14     //declare array
15     double sales[4] = {0.0};
16
17     //fill array with data
18     for (int sub = 0; sub < 4; sub += 1)
19     {
20           cout << "Enter the sales for Region ";
21           cout << sub + 1 << ": ";
22           cin >> sales[sub];
23     }     //end for
24
25     //display the contents of the array
26     displayArray(sales, 4);
27
28     system("pause");
29     return 0;
30 }  //end of main function
31
32 //*****function definitions*****
33 void displayArray(double dollars[], int numElements)
34 {
35     cout << fixed << setprecision(2) << endl << endl;
36     for (int sub = 0; sub < numElements; sub += 1)
37     {
38           cout << "Sales for Region " << sub + 1 << ": $";
39           cout << dollars[sub] << endl;
40     }     //end for
41 }  //end of displayArray function
```

the name is optional

function prototype

function call

your C++ development tool may not require this statement

function header

Figure 11-19 XYZ Company's modified sales program

Figure 11-20 shows the completed desk-check table for the XYZ Company's modified sales program. Recall from Chapter 10 that when you pass a variable *by reference* to a function, the computer locates the variable and then assigns the name of the corresponding formal parameter to the variable. The same process occurs with array variables and explains why each array variable in Figure 11-20 has two names: one assigned by the `main` function, and the other assigned by the `displayArray` function. Although both functions can access the memory locations where the array variables reside, each function uses a different name to do so. The `main` function uses the name `sales`, whereas the `displayArray` function uses the name `dollars`.

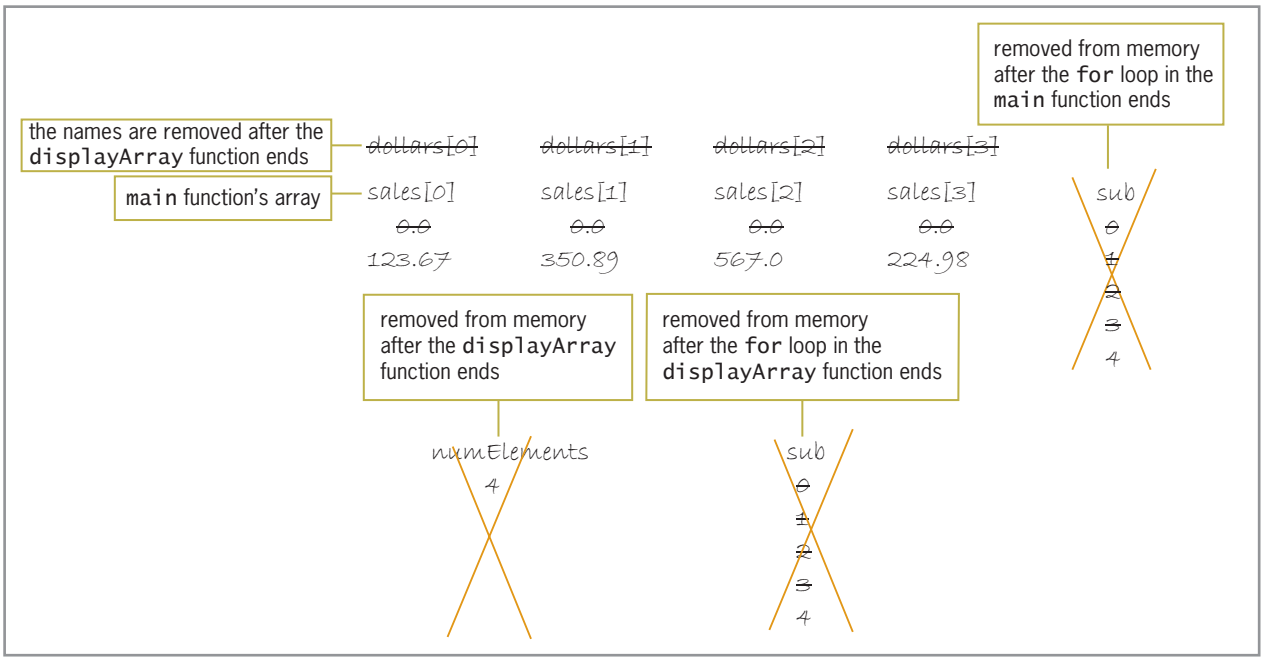


Figure 11-20 Completed desk-check table for the XYZ Company's modified sales program

Mini-Quiz 11-1



The answers to Mini-Quiz questions are located in Appendix A.

1. Which of the following declares a one-dimensional `int` array named `quantities` and initializes each of its 20 elements to the number 0?
   a. `int quantities[20] = {0};`
   b. `int quantities(20) = {0};`
   c. `int quantities{20} = 0;`
   d. none of the above

2. What is the name of the first element in the `quantities` array from Question 1?

3. What is the name of the last element in the `quantities` array from Question 1?

4. Write a C++ statement that assigns the number 7 to the fourth element in the `quantities` array from Question 1.

5. Which of the following calls the `getTotal` function, passing it the `quantities` array from Question 1 and the number of array elements? The `getTotal` function is a value-returning function.
   a. `total = getTotal(quantities[], 20);`
   b. `total = getTotal(quantities[20]);`
   c. `total = getTotal(quantities, 20);`
   d. none of the above

The Moonbucks Coffee Program—Calculating a Total and Average

Figure 11-21 shows the problem specification, IPO chart information, and C++ instructions for the Moonbucks Coffee program. (The flowchart for this program is contained in the Ch11Flowcharts.pdf file, which is located in the Cpp6\Chap11 folder.) The program displays both the total and average number of pounds of coffee used in a 12-month period. Notice that the program stores the monthly usage amounts in a 12-element `double` array named `pounds`. It also uses a program-defined value-returning function named `getTotal` to calculate the total usage for the year. The `getTotal` function does this by adding together each monthly value stored in the `pounds` array, which is passed to the function when it is invoked. The `getTotal` function then returns the sum to the `main` function, where it is used to calculate the average usage.

Problem specification
The store manager at Moonbucks Coffee wants a program that displays both the total and average number of pounds of coffee used during a 12-month period. Last year, the pounds of coffee used each month were as follows: 400.5, 450, 475.5, 336.5, 457, 325, 220.5, 276, 300, 320.5, 400.5, 415. The program will use two value-returning functions: `main` and `getTotal`. The `getTotal` function will calculate the total number of pounds of coffee used.

`main` function

IPO chart information	**C++ instructions**
Input	
array (12 elements)	`double pounds[12] = {400.5, 450.0, 475.5, 336.5, 457.0, 325.0, 220.5, 276.0, 300.0, 320.5, 400.5, 415.0};`
Processing	
none	
Output	
total pounds	`double total = 0.0;`
average pounds	`double average = 0.0;`
Algorithm	
1. call the getTotal function to calculate the total pounds, pass the function the pounds array and the number of elements in the array	`total = getTotal(pounds, 12);`
2. calculate the average pounds by dividing the total pounds by the number of array elements	`average = total / 12;`
3. display the total pounds and average pounds	`cout << "Total pounds: " << total << endl;` `cout << "Average pounds: " << average << endl;`

Figure 11-21 Problem specification, IPO chart information, and C++ instructions for the Moonbucks Coffee program *(continues)*

(continued)

getTotal function

IPO chart information	C++ instructions
Input	
array	`double poundsUsed[]` (formal parameter)
number of elements in the array	`int numElements` (formal parameter)
Processing	
subscript counter (0 to 11)	declared and initialized in the for clause
Output	
total pounds (accumulator)	`double totalUsed = 0.0;`
Algorithm	
1. repeat for (each array element)	`for (int sub = 0; sub < numElements; sub += 1)`
add the current array element's value to the total pounds	`totalUsed += poundsUsed[sub];`
end repeat	`//end for`
2. return the total pounds	`return totalUsed;`

Figure 11-21 Problem specification, IPO chart information, and C++ instructions for the Moonbucks Coffee program

Figure 11-22 shows the entire Moonbucks Coffee program. The statement on Lines 15 through 17 in the `main` function declares the 12-element `double` `pounds` array. It uses the 12 values provided in the problem specification to initialize the array. The statements on Lines 19 and 20 declare two `double` variables named `total` and `average`. Next, the assignment statement on Line 23 calls the `getTotal` function, passing it the `pounds` array and the number of elements in the array. Recall that when an array is passed to a function, the computer passes only the address of the first array element. At this point, the computer temporarily leaves the `main` function to process the `getTotal` function's code, beginning with the function header on Line 35. When processing the function header, the computer locates the `pounds` array in memory and assigns the formal parameter's name—in this case, `poundsUsed`—to each element. As a result, each array element has two names. The first element, for example, is called `pounds[0]` in the `main` function but `poundsUsed[0]` in the `getTotal` function. After processing the `getTotal` function's header, the computer processes the statements within the function body. Those statements use a `for` loop, along with an accumulator variable named `totalUsed`, to add together each value contained in the `poundsUsed` array. The sum of those values represents the total number of pounds of coffee used. The `getTotal` function's `return` statement returns the total number of pounds to the assignment statement on Line 23 in the `main` function. At that point, the `getTotal` function ends and the computer removes the `poundsUsed` name from the array elements. It also removes the `numElements` and `totalUsed` variables from internal memory. The assignment statement on Line 23 assigns the total number of pounds to the `main` function's `total` variable. Next, the assignment statement on Line 24 is processed. That statement calculates the average number



The `sub` variable is removed when the `for` loop in the `getTotal` function ends.

of pounds by dividing the total number of pounds by the number of array elements (12). It then assigns the result to the `average` variable. The `cout` statements on Lines 27 and 28 display the total number of pounds used and the average number of pounds used, respectively. The computer then processes the `system("pause");` and `return 0;` statements on Lines 30 and 31. When the program ends, the computer removes the `pounds` array, as well as the `total` and `average` variables, from internal memory. Figure 11-23 shows the program's output.

```
1  //Moonbucks Coffee.cpp
2  //Displays the total and average number of pounds
3  //of coffee used during a 12-month period
4  //Created/revised by <your name> on <current date>
5
6  #include <iostream>
7  using namespace std;
8
9  //function prototype
10 double getTotal(double poundsUsed[], int numElements);
11
12 int main()
13 {
14    //declare array
15    double pounds[12] = {400.5, 450.0,
16           475.5, 336.5, 457.0, 325.0, 220.5,
17           276.0, 300.0, 320.5, 400.5, 415.0};
18    //declare variables
19    double total   = 0.0;
20    double average = 0.0;
21
22    //calculate the total and average pounds used
23    total = getTotal(pounds, 12);
24    average = total / 12;
25
26    //display the total and average pounds used
27    cout << "Total pounds: " << total << endl;
28    cout << "Average pounds: " << average << endl;
29
30    system("pause");
31    return 0;
32 }  //end of main function
33
34 //*****function definitions*****
35 double getTotal(double poundsUsed[], int numElements)
36 {
37    double totalUsed = 0.0;     //accumulator
38
39    //accumulate the pounds used
40    for (int sub = 0; sub < numElements; sub += 1)
41         totalUsed += poundsUsed[sub];
42    //end for
43
44    return totalUsed;
45 }  //end of getTotal function
```

your C++ development tool may not require this statement



The `for` loop in Figure 11-22 will end when the `sub` variable contains the integer 12, because that is the first integer that is not less than 12.

Figure 11-22 Moonbucks Coffee program

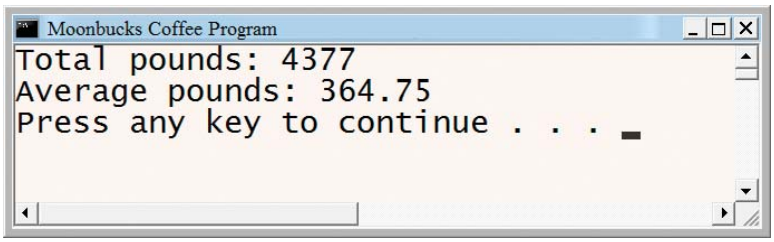


Figure 11-23 Result of running the Moonbucks Coffee program

The KL Motors Program—Searching an Array

Figure 11-24 shows the problem specification, IPO chart information, and C++ instructions for the KL Motors program. (The flowchart for this program is contained in the Ch11Flowcharts.pdf file, which is located in the Cpp6\Chap11 folder.) The program displays the number of employees whose salary is greater than the amount entered by the user. Notice that the program stores each employee's salary in a 10-element `int` array named `salaries`. To accomplish its task, the program will use a loop to search the `salaries` array and use a selection structure to compare the salary in the current array element with the salary entered by the user. If the salary in the current array element is greater than the one entered by the user, the program adds the number 1 to the `numEarnOver` counter variable. After searching each array element, the program displays the contents of the `numEarnOver` variable on the screen. It then prompts the user to enter another salary amount.

Problem specification
The payroll manager at KL Motors wants a program that displays the number of employees who earn more than a specific amount, which he will enter. The company employs 10 people. Their annual salaries are as follows: 23000, 26000, 34000, 21000, 54000, 45000, 36000, 80000, 75000, 34000. The program will use only the `main` function. It will use a sentinel value to end the program.

IPO chart information	**C++ instructions**
Input	
array (10 elements)	`int salaries[10] = {23000, 26000, 34000, 21000, 54000, 45000, 36000, 80000, 75000, 34000};`
salary to search for	`int searchFor = 0;`
Processing	
subscript counter (0 to 9)	*declared and initialized in the for clause*
Output	
number earning over the salary to search for (counter)	`int numEarnOver = 0;`

Figure 11-24 Problem specification, IPO chart information, and C++ instructions for the KL Motors program *(continues)*

(continued)

Algorithm

```
1. enter the salary to search for
                                        cout << "Salary to search for "
                                        << "(negative number to end): ";
                                        cin >> searchFor;
2. repeat while (the salary to search   while (searchFor >= 0)
   for is greater than or equal to 0)   {

      repeat for (each array element)       for (int sub = 0; sub < 10; sub += 1)
         if (the current array element's       if (salaries[sub] > searchFor)
         value is greater than the salary
         to search for)
            add 1 to the number earning             numEarnOver += 1;
            over the salary to search for
         end if                                 //end if
      end repeat                            //end for

      display the number earning over       cout << "Number of employees earning "
      the salary to search for              << "more than $" << searchFor << ": "
                                            << numEarnOver << endl;

      enter the salary to search for        cout << "Salary to search for "
                                            << "(negative number to end): ";
                                            cin >> searchFor;
      reset the number earning over         numEarnOver = 0;
      the salary to search for
   end repeat                           }  //end while
```

Figure 11-24 Problem specification, IPO chart information, and C++ instructions for the KL Motors program

Figure 11-25 shows the entire KL Motors program. The declaration statement on Lines 11 through 15 creates the 10-element `salaries` array and initializes it using the 10 values provided in the problem specification. The statements on Lines 16 and 17 declare two `int` variables named `searchFor` and `numEarnOver`. The `cout` statement on Lines 20 and 21 prompts the user to enter a salary amount, and the `cin` statement on Line 22 stores the user's response in the `searchFor` variable. The `while` loop in the program repeats its instructions as long as the value in the `searchFor` variable is greater than or equal to the number 0. Within the `while` loop is a nested `for` loop, which the program uses to access each element in the `salaries` array, beginning with the element whose subscript is 0 and ending with the element whose subscript is 9. The selection structure in the nested loop compares the salary stored in the current array element with the salary stored in the `searchFor` variable. If the array element contains a salary that is greater than the salary stored in the `searchFor` variable, the selection structure's true path adds the number 1 to the contents of the `numEarnOver` variable. In the program, the `numEarnOver` variable is used as a counter variable to keep track of the number of employees earning more than the amount entered by the user. When the nested `for` loop ends, which is when the `sub` variable contains the number 10, the `cout` statement on Line 34 through 36 displays an appropriate message on the screen. As shown in Figure 11-26, the `cout` statement displays the message "Number of employees earning more than

$35000: 5" when the payroll manager enters 35000. The statements on Lines 39 through 41 then prompt the user to enter another salary amount and store the user's response in the `searchFor` variable. The statement on Line 42 resets the `numEarnOver` counter variable. Next, the condition in the `while` clause on Line 24 is evaluated again. If the `searchFor` variable's value is greater than or equal to the number 0, the instructions in the `while` loop are processed again. Otherwise, the loop ends, and the computer processes the statements on Lines 44 and 45 before the program ends.

```
 1  //KL Motors.cpp - displays the number of employees
 2  //whose salary is greater than a specific amount
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  using namespace std;
 7
 8  int main()
 9  {
10     //declare array and variables
11     int salaries[10] = {23000, 26000,
12                         34000, 21000,
13                         54000, 45000,
14                         36000, 80000,
15                         75000, 34000};
16     int searchFor   = 0;
17     int numEarnOver = 0;     //counter
18
19     //get salary to search for
20     cout << "Salary to search for "
21          << "(negative number to end): ";
22     cin >> searchFor;
23
24     while (searchFor >= 0)
25     {
26          //search the array
27          for (int sub = 0; sub < 10; sub += 1)
28               if (salaries[sub] > searchFor)
29                    numEarnOver += 1;
30               //end if
31          //end for
32
33          //display the search results
34          cout << "Number of employees earning "
35          << "more than $" << searchFor << ": "
36          << numEarnOver << endl;
37
38          //get another salary to search for
39          cout << "Salary to search for "
40               << "(negative number to end): ";
41          cin >> searchFor;
42          numEarnOver = 0;
43     }    //end while
44     system("pause");
45     return 0;
46  }  //end of main function
```

your C++ development tool may not require this statement

Figure 11-25 KL Motors program

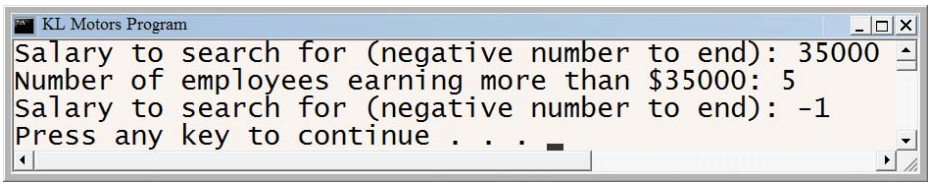

KL Motors Program
Salary to search for (negative number to end): 35000
Number of employees earning more than $35000: 5
Salary to search for (negative number to end): -1
Press any key to continue . . .

Figure 11-26 Sample run of the KL Motors program

The Hourly Rate Program—Accessing an Individual Element

Figure 11-27 shows the problem specification, IPO chart information, and C++ instructions for the hourly rate program. (The flowchart for this program is contained in the Ch11Flowcharts.pdf file, which is located in the Cpp6\Chap11 folder.) The program uses a six-element array to store the hourly rates, each of which is associated with a specific pay code. The program prompts the user to enter a pay code and then determines whether the pay code is valid. To be valid, the pay code must be greater than or equal to the number 1 and (at the same time) less than or equal to the number 6. If the pay code is valid, the program uses the pay code to display the appropriate hourly rate from the array. If the pay code is not valid, the program displays the message "Invalid pay code".

Problem specification
Create a program that displays the hourly rate associated with the pay code entered by the user. The pay codes and hourly rates are shown here:

Pay code	Hourly rate
1	11.25
2	10.00
3	9.85
4	8.65
5	15.00
6	25.00

IPO chart information	**C++ instructions**
Input	
pay code	`int code = 0;`
Processing	
array [6 elements]	`double hourlyRates[6] = {11.25, 10.0, 9.85, 8.65, 15.0, 25.0};`
Output	
hourly rate	displayed from the array

Figure 11-27 Problem specification, IPO chart information, and C++ instructions for the hourly rate program *(continues)*

(continued)

Algorithm

Algorithm	C++
1. enter the pay code	`cout << "Pay code (1 - 6): ";` `cin >> code;`
2. if (the pay code is greater than or equal to 1 and less than or equal to 6)	`if (code >= 1 && code <= 6)`
display the hourly rate from the array, using the pay code minus 1 as the subscript	`cout << "Hourly rate: $" <<` `hourlyRates[code - 1] << endl;`
else	`else`
display "Invalid pay code" message	`cout << "Invalid pay code" << endl;`
end if	`//end if`

Figure 11-27 Problem specification, IPO chart information, and C++ instructions for the hourly rate program

Figure 11-28 shows the entire hourly rate program. The declaration statement on Lines 12 and 13 creates the six-element `hourlyRates` array and initializes it using the hourly rates provided in the problem specification. The rate associated with pay code 1 is stored in the first array element (`hourlyRates[0]`). The rate associated with pay code 2 is stored in the second array element `[hourlyRates[1]`), and so on. Notice that the pay code is one number more than the subscript of its corresponding hourly rate in the array. The statement on Line 14 in the program declares and initializes an `int` variable named `code`. The `code` variable will store the pay code entered by the user. It also will be used to access the corresponding hourly rate in the array. The statement on Line 17 is processed next and tells the computer to display the hourly rate in fixed-point notation with two decimal places. The `cout` statement on Line 20 prompts the user to enter the pay code, and the `cin` statement on Line 21 stores the user's response in the `code` variable. Next, the `if` clause on Line 22 determines whether the pay code stored in the `code` variable is valid. If the pay code is valid, the instruction in the `if` statement's true path displays the appropriate hourly rate from the array; otherwise, the instruction in the `if` statement's false path displays the message "Invalid pay code". Study carefully the instruction in the `if` statement's true path; the instruction appears on Lines 23 and 24 in the program. Notice that the instruction uses the `code` variable, which contains the pay code entered by the user, to access the appropriate element in the `hourlyRates` array. Also notice that, to access the correct element, the number 1 must be subtracted from the contents of the `code` variable. This is because the pay code stored in the variable is one number more than the subscript of its associated hourly rate in the array. Figure 11-29 shows a sample run of the hourly rate program.



Before accessing an array element, you always should verify that the subscript is valid for the array. If the compiler encounters an invalid subscript, it will display an error message and the program will end abruptly.

```
 1  //Hourly Rate.cpp - displays the hourly rate
 2  //associated with the pay code entered by the user
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  #include <iomanip>
 7  using namespace std;
 8
 9  int main()
10  {
11     //declare array and variable
12     double hourlyRates[6] = {11.25, 10.0, 9.85,
13                              8.65, 15.0, 25.0};
14     int code = 0;
15
16     //display hourly rate with two decimal places
17     cout << fixed << setprecision(2);
18
19     //get pay code
20     cout << "Pay code (1 - 6): ";
21     cin >> code;
22     if (code >= 1 && code <= 6)
23          cout << "Hourly rate: $" <<
24               hourlyRates[code - 1] << endl;
25     else
26          cout << "Invalid pay code" << endl;
27     //end if
28
29     system("pause");
30     return 0;
31  }  //end of main function
```

your C++ development tool may not require this statement

Figure 11-28 Hourly rate program

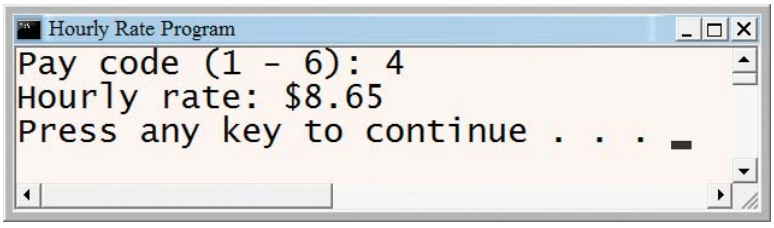


Figure 11-29 Sample run of the hourly rate program

The Random Numbers Program

Figure 11-30 shows the problem specification, IPO chart information, and C++ instructions for the random numbers program. (The flowchart for this program is contained in the Ch11Flowcharts.pdf file, which is located in the Cpp6\Chap11 folder.) The `main` function assigns random numbers from 1 through 100 to a five-element `int` array named `randNums`. It then calls a program-defined void function to display the contents of the array. Next, it calls a program-defined value-returning function to determine the highest number in the array. The `main` function displays the function's return value on the screen.

Problem specification
Create a program that assigns random integers from 1 through 100 to a five-element array. The program should display both the contents of the array and the highest number stored in the array. Use a program-defined void function to display the array's contents. Use a program-defined value-returning function to determine the highest number in the array.

`main` function

IPO chart information	C++ instructions
Input	
random number (5 from 1 to 100)	generated by the program and stored in the array
Processing	
array (5 elements)	`int randNums[5] = {0};`
subscript counter (0 to 4)	declared and initialized in the for clause
Output	
random number (5 from 1 to 100)	displayed by the displayArray function
highest value in the array	determined by the getHighest function
Algorithm	
1. initialize the random number generator	`srand(static_cast<int>(time(0)));`
2. repeat for (subscript counter from 0 to 4)	`for (int sub = 0; sub < 5; sub += 1)`
generate a random number and store it in the current array element	`randNums[sub] = 1 + rand() % (100 - 1 + 1);`
end repeat	`//end for`
3. call the displayArray function to display the contents of the array, pass the array and the number of elements	`displayArray(randNums, 5);`
4. call the getHighest function to determine the highest number in the array, pass the array and the number of elements, then display the highest number	`cout << endl << "Highest number: " << getHighest(randNums, 5) << endl;`

`displayArray` function

IPO chart information	C++ instructions
Input	
array (5 elements)	`int numbers[]` (formal parameter)
number of elements	`int numElements` (formal parameter)
Processing	
subscript counter (0 to the number of elements)	declared and initialized in the for clause
Output	
array (5 elements)	displayed from the array by the for loop

Figure 11-30 Problem specification, IPO chart information, and C++ instructions for the random numbers program *(continues)*

(continued)

Algorithm	

repeat for (subscript counter from 0 to the number of elements)
  display the contents of the current array element
end repeat
	`for (int sub = 0; sub < numElements; sub += 1)`
	`  cout << numbers[sub] << endl;`
	
	`//end for`

getHighest function **IPO chart information**	**C++ instructions**
Input	
array (5 elements) number of elements	`int numbers[]` (formal parameter) `int numElements` (formal parameter)
Processing	
subscript counter	`int x = 1;`
Output	
highest number	`int high = numbers[0];`
Algorithm	

1. repeat while (the subscript counter is less than the number of elements)
  if (the current array element's value is greater than the highest number)
  assign the current array element's value as the highest number
  end if
  add 1 to the subscript counter
  end repeat
	`while (x < numElements)`
	`{`
	
	`  if (numbers[x] > high)`
	
	`  high = numbers[x];`
	
	`  //end if`
	`  x += 1;`
	`}  //end while`

2. return highest number	`return high;`

Figure 11-30 Problem specification, IPO chart information, and C++ instructions for the random numbers program

Figure 11-31 shows a sample run of the random numbers program, and Figure 11-32 shows all of the program's code. The program contains two program-defined value-returning functions: `main` and `getHighest`. It also contains a program-defined void function named `displayArray`.

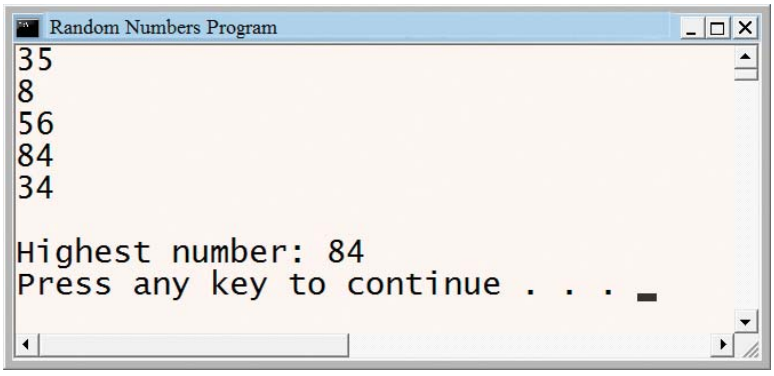


Figure 11-31 Random numbers program

```
 1  //Random Numbers.cpp - displays the highest
 2  //random number stored in an array
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  #include <ctime>
 7  using namespace std;
 8
 9  //function prototypes
10  void displayArray(int numbers[], int numElements);
11  int getHighest(int numbers[], int numElements);
12
13  int main()
14  {
15     //declare array
16     int randNums[5] = {0};
17
18     //initialize random number generator
19     srand(static_cast<int>(time(0)));
20     //assign random integers from 1
21     //through 100 to the array
22     for (int sub = 0; sub < 5; sub += 1)
23          randNums[sub] = 1 + rand() % (100 - 1 + 1);
24     //end for
25
26     //display array
27     displayArray(randNums, 5);
28
29     //display highest number in the array
30     cout << endl << "Highest number: "
31          << getHighest(randNums, 5) << endl;
32
33     system("pause");
34     return 0;
35  }  //end of main function
36
37  //*****function prototype*****
38  void displayArray(int numbers[], int numElements)
39  {
40     for (int sub = 0; sub < numElements; sub += 1)
41          cout << numbers[sub] << endl;
42     //end for
43  }  //end of displayArray function
44
45  int getHighest(int numbers[], int numElements)
46  {
47     //assign first element's value
48     //to the high variable
49     int high = numbers[0];
50
51     //begin the search with the second element
52     int x = 1;
53
```

your C++ development tool may not require this statement



The loop in the `getHighest` function searches the second through the last elements in the `numbers` array. It doesn't need to search the first element because that element's value is already stored in the `high` variable.

Figure 11-32 Random numbers program *(continues)*

(continued)

```
54      //search for highest number
55      while (x < numElements)
56      {
57           if (numbers[x] > high)
58                high = numbers[x];
59           //end if
60           x += 1;
61      }    //end while
62
63      return high;
64 }    //end of getHighest function
```

Figure 11-32 Random numbers program

Desk-checking the program will help you understand how the highest number is determined. The statement on Line 16 in the program declares and initializes a five-element `int` array named `randNums`. The statement on Line 19 initializes the C++ random number generator, and the `for` loop on Lines 22 through 24 assigns five random integers to the `randNums` array. Figure 11-33 shows the desk-check table after the `for` loop ends, assuming the random numbers are 35, 8, 56, 84, and 34.

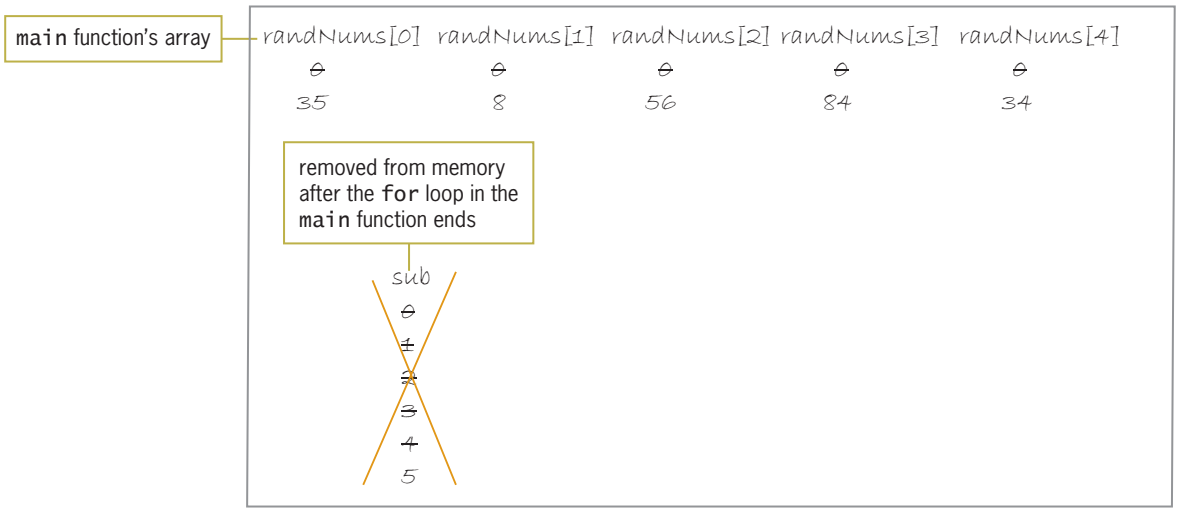


Figure 11-33 Desk-check table after the `for` loop on Lines 22 through 24 ends

The statement on Line 27 calls the void `displayArray` function, passing it two actual arguments: the `randNums` array and the number of array elements (5). At this point, the computer temporarily leaves the `main` function to process the `displayArray` function's code, beginning with the function header on Line 38. When processing the function header, the computer locates the `randNums` array in memory and assigns the name of the first formal parameter (`numbers`) to each element. It also creates an `int` variable named `numElements` and stores the integer 5 in it. Figure 11-34 shows the desk-check table after the `displayArray` function header is processed.

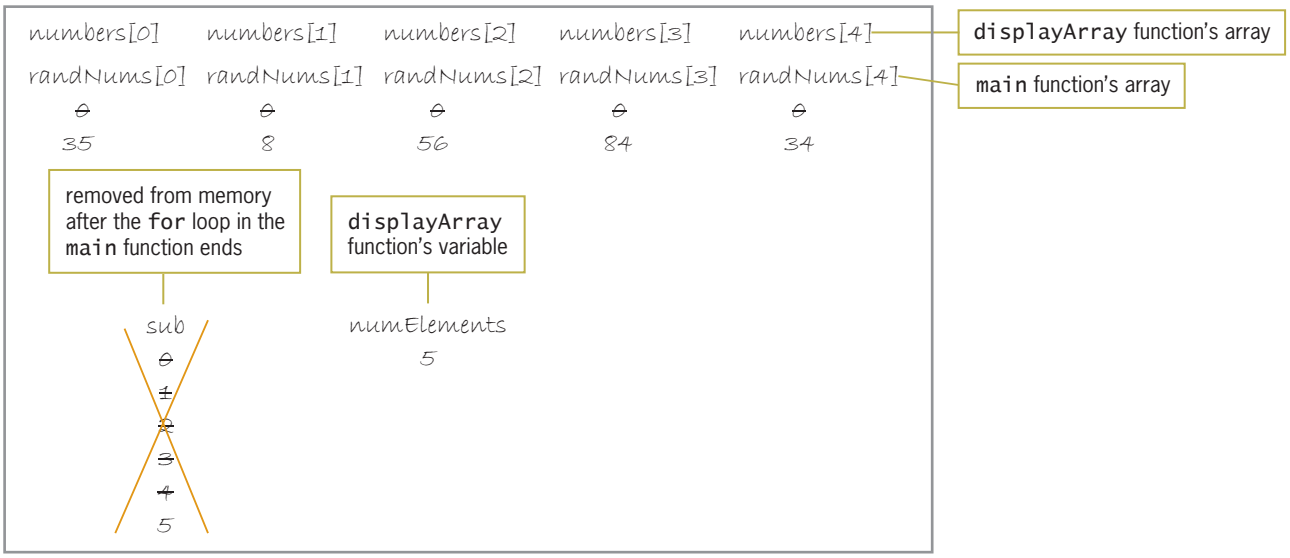


Figure 11-34 Desk-check table after the `displayArray` function header is processed

The `displayArray` function displays the contents of the `numbers` array (which also is the `randNums` array) on the screen. Figure 11-35 shows the desk-check table after the `displayArray` function ends.

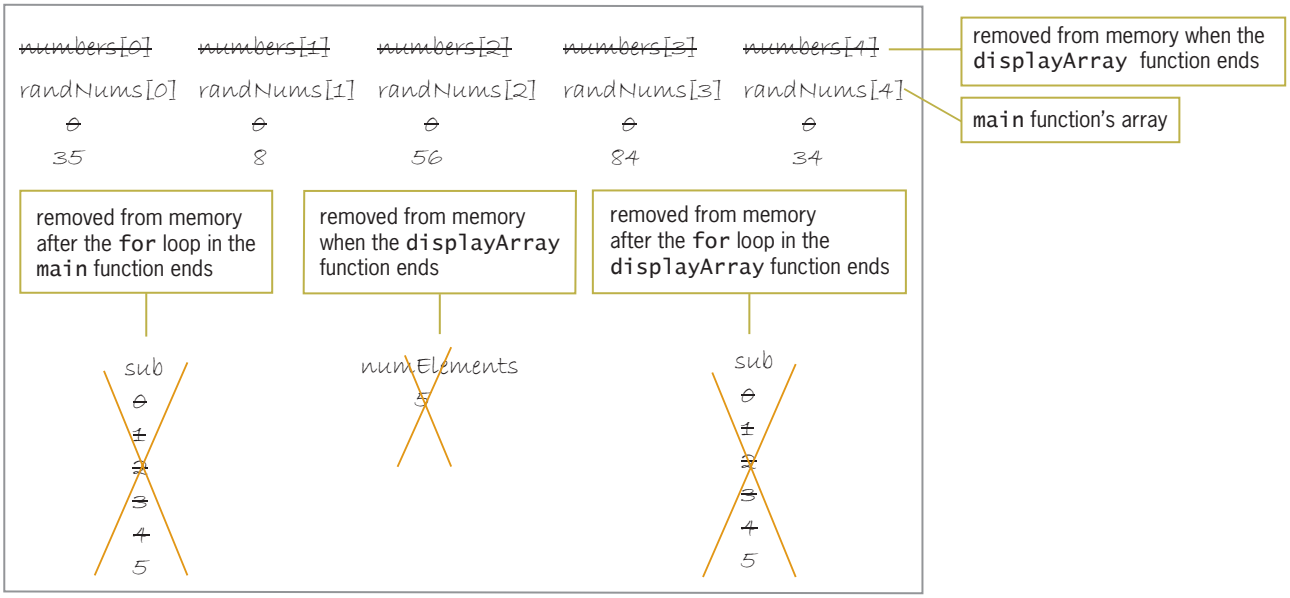


Figure 11-35 Desk-check table after the `displayArray` function ends

After processing the `displayArray` function, the computer returns to the `main` function to process the `cout` statement that appears on Lines 30 and 31. The statement calls the value-returning `getHighest` function, passing it two actual arguments: the `randNums` array and the number of array elements (5). At this point, the computer temporarily leaves the `main` function to process the `getHighest` function's code, beginning with the function header on Line 45. When processing the function header, the computer locates the `randNums` array in memory and assigns the name of the first

formal parameter—numbers—to each array element. It also creates the second formal parameter—an int variable named numElements—and stores the number 5 in it. Next, the declaration statements on Lines 49 and 52 create two int variables named high and x. The high variable will keep track of the highest value stored in the array, and the x variable will keep track of the array subscripts. Notice that the high variable is initialized to the value stored in the first array element (35). The x variable is initialized to the number 1, which is the subscript of the second array element. Figure 11-36 shows the desk-check table after the declaration statements are processed.

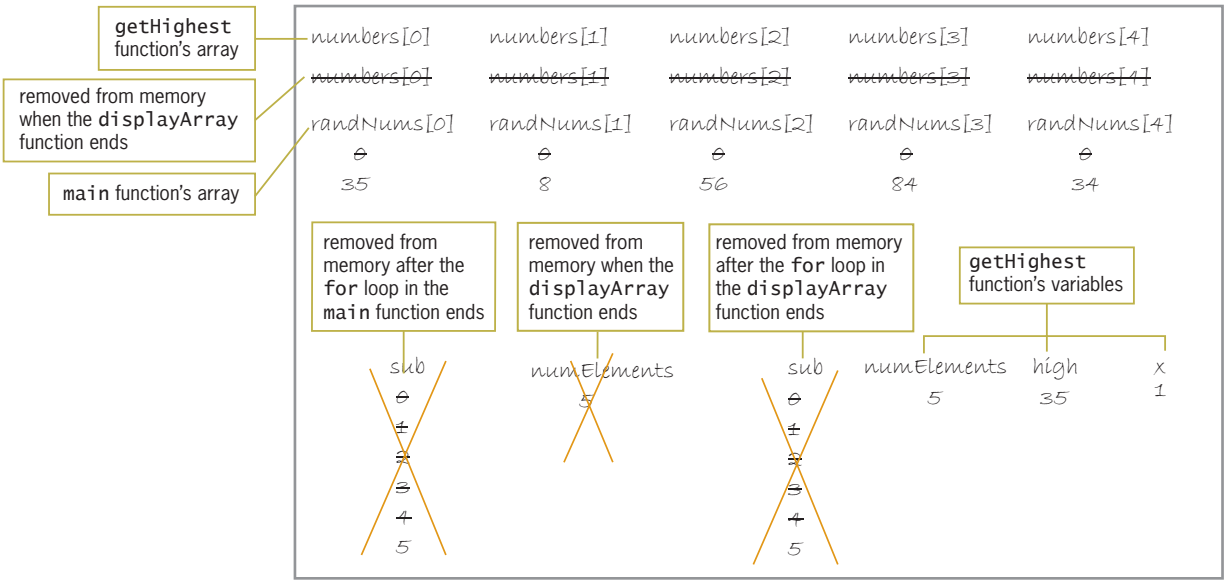


Figure 11-36 Desk-check table after the declaration statements on Lines 49 and 52 are processed

The while clause on Line 55 is processed next. The clause's condition checks whether the x variable's value is less than the number of elements stored in the numElements variable. It is, so the if statement's condition compares the value stored in the numbers[1] element, which is the second element in the array, with the value stored in the high variable. (Recall that at this point, the high variable contains the same value as the first array element.) The value in the numbers[1] element (8) is not greater than the value in the high variable (35), so the if statement ends and the computer processes the x += 1; statement on Line 60. The statement increases the value in the x variable by 1, giving 2. Next, the while clause's condition checks whether the x variable's value is less than the value stored in the numElements variable. It is, so the if statement's condition compares the value stored in the numbers[2] element with the value stored in the high variable. The value in the numbers[2] element (56) is greater than the value in the high variable (35), so the instruction in the if statement's true path assigns the element's value to the high variable, as shown in Figure 11-37.

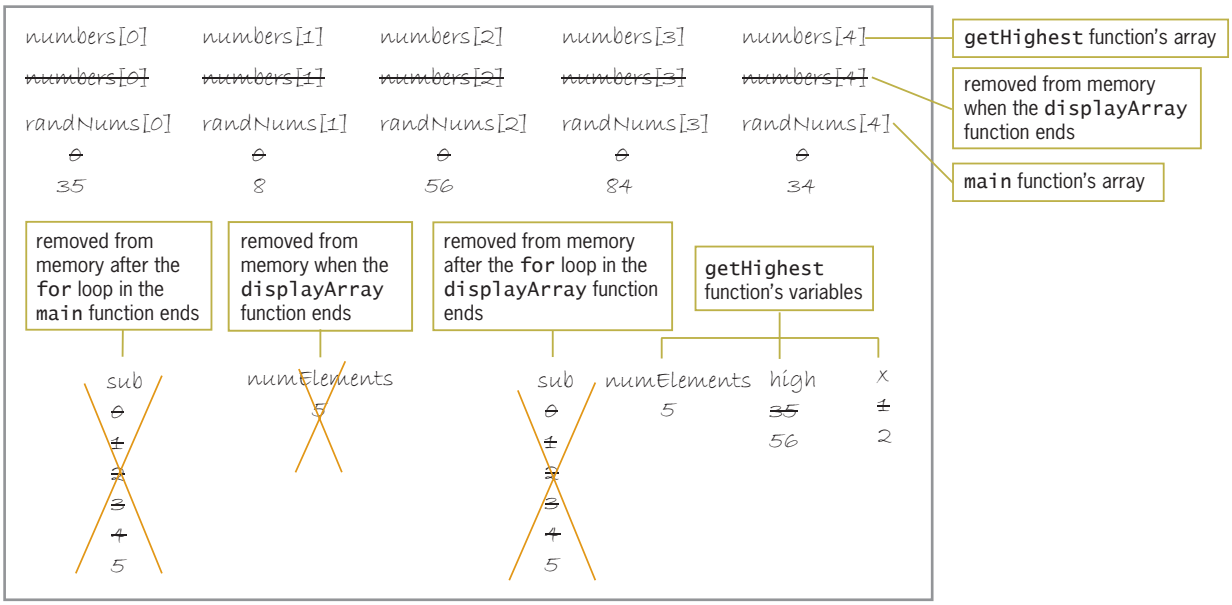


Figure 11-37 Desk-check table showing the third element's value entered in the `high` variable

Next, the computer processes the `x += 1;` statement on Line 60. The statement increases the value in the `x` variable by 1, giving 3. The `while` clause on Line 55 is processed next. The clause's condition checks whether the `x` variable's value is less than the number of elements stored in the `numElements` variable. It is, so the `if` statement's condition compares the value stored in the `numbers[3]` element with the value stored in the `high` variable. The value in the `numbers[3]` element (84) is greater than the value in the `high` variable (56), so the instruction in the `if` statement's true path assigns the element's value to the `high` variable, as shown in Figure 11-38.

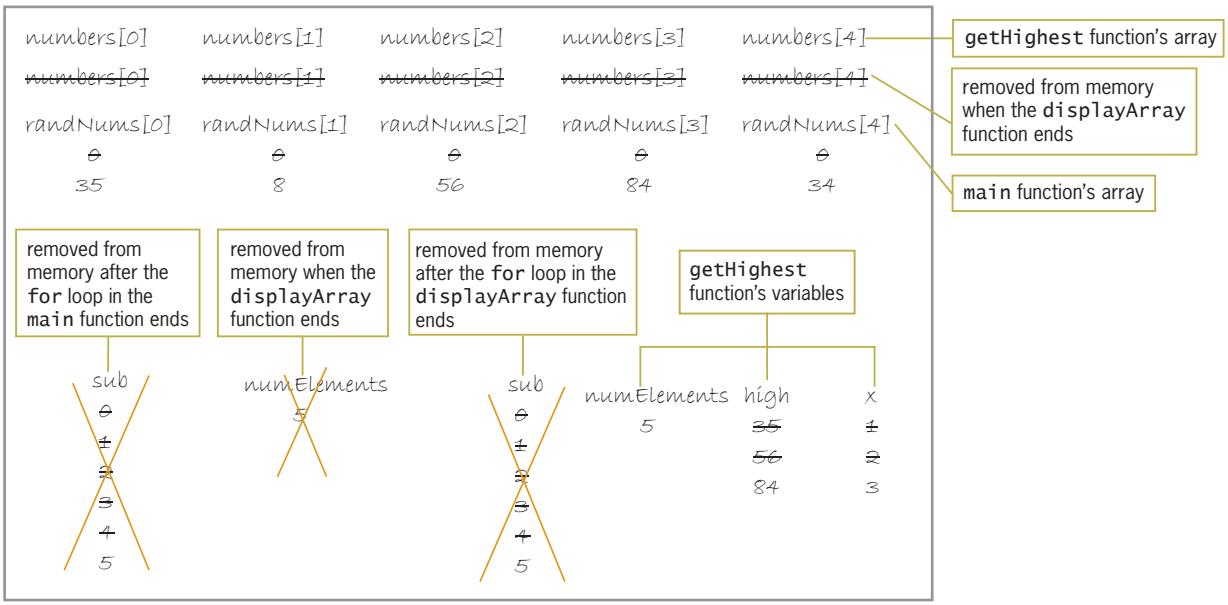


Figure 11-38 Desk-check table showing the fourth element's value entered in the `high` variable

Next, the computer processes the `x += 1;` statement on Line 60. The statement increases the value in the `x` variable by 1, giving 4. The condition in the `while` clause on Line 55 checks whether the `x` variable's value is less than the number of elements stored in the `numElements` variable. It is, so the `if` statement's condition compares the value stored in the `numbers[4]` element with the value stored in the `high` variable. The value in the `numbers[4]` element (34) is not greater than the value in the `high` variable (84), so the `if` statement ends and the computer processes the `x += 1;` statement on Line 60. The statement increases the value in the `x` variable by 1, giving 5. Next, the `while` clause's condition checks whether the `x` variable's value is less than the value stored in the `numElements` variable. It's not, so the `while` loop ends and the computer processes the `return` statement on Line 63. The statement returns the value stored in the `high` variable to the statement that called the `getHighest` function. That statement is the `cout` statement that appears on Lines 30 and 31 in the `main` function. After the `getHighest` function ends, the computer removes the `numbers` name from each element in the array. It also removes the `numElements`, `high`, and `x` variables from memory. The completed desk-check table is shown in Figure 11-39.

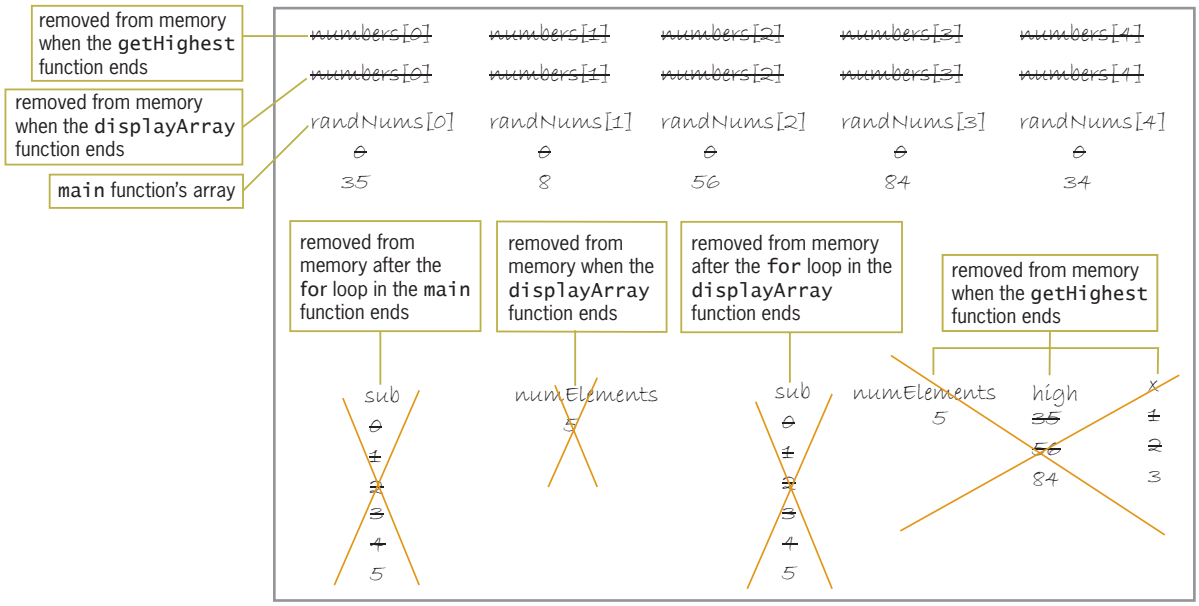


Figure 11-39 Completed desk-check table for the random numbers program

The `cout` statement on Lines 30 and 31 in the `main` function displays the value returned by the `getHighest` function. The computer then processes the `system("pause");` and `return 0;` statements before the program ends. When the program ends, the computer removes the `randNums` array from memory.

Mini-Quiz 11-2



The answers to Mini-Quiz questions are located in Appendix A.

1. Which of the following increases the `total` variable by the contents of the third element in the `orders` array? The `total` variable and the `orders` array have the `int` data type.

   a. `orders[2] += total;`

   b. `orders[3] += total;`

   c. `total += orders[2];`

   d. `total += orders[3];`

2. Which of the following `if` clauses determines whether the value stored in the fourth element in the `orders` array is greater than 25? The array has the `int` data type.

   a. `if (orders(3) > 25)`

   b. `if (orders{4} > 25)`

   c. `if (orders[3] > 25)`

   d. `if (orders[4] > 25)`

3. Write a C++ statement that multiplies the contents of the first element in the `sales` array by .15 and then stores the result in the `bonus` variable. The `sales` array and `bonus` variable have the `double` data type.

4. Which of the following `if` clauses determines whether an `int` variable named `sub` contains a valid subscript for the `scores` array? The array has 10 elements.

   a. `if (sub > 0 && sub < 10)`

   b. `if (sub >= 0 && sub <= 10)`

   c. `if (sub >= 0 && sub < 10)`

   d. `if (sub > 0 && sub <= 10)`

5. Which of the following `while` clauses tells the computer to process the loop instructions for each of the 20 elements in the `inventory` array? The program uses an `int` variable named `x` to keep track of the array subscripts. The `x` variable is initialized to 0.

   a. `while (x < 20)`

   b. `while (x <= 20)`

   c. `while (x > 0)`

   d. `while (x >= 0)`

Sorting the Data Stored in a One-Dimensional Array

In some programs, you might need to arrange the contents of a one-dimensional array in either ascending or descending order. Arranging data in a specific order is called **sorting**. When a one-dimensional array is sorted in ascending order, the first element in the array contains the smallest value and the last element contains the largest value. When a one-dimensional array is sorted in descending order, on the other hand, the first element contains the largest value and the last element contains the smallest value. Over the years, many different sorting algorithms have been developed; one such algorithm is called the bubble sort. The **bubble sort** provides a quick and easy way to sort the items stored in an array, as long as the number of items is relatively small—for example, fewer than 50. The bubble sort algorithm works by comparing adjacent array elements and interchanging (swapping) the ones that are out of order. The algorithm continues comparing and swapping until the data in the array is sorted. To demonstrate the logic of a bubble sort, you will manually sort the contents of a three-element array in ascending order. The array, which is named `nums`, contains the following numbers: 9, 8, and 7. Figure 11-40 shows the `nums` array values before, during, and after the bubble sort. The bubble sort algorithm begins by comparing the first value in the array with the second value. If the first value is less than or equal to the second value, then no swap is made. However, if the first value is greater than the second value, then both values are interchanged. In this case, the first value (9) is greater than the second value (8), so the values are swapped as shown in the Second Comparison column in Figure 11-40. After comparing the first value in the array with the second value, the algorithm then compares the second value with the third value. In this case, 9 is greater than 7, so the two values are swapped as shown in the Result column in Figure 11-40. At this point, the algorithm has completed its first time through the entire array—referred to as a pass. Notice that at the end of the first pass, the largest value (9) is stored in the last element in the array. The bubble sort gets its name from the fact that as the larger values drop to the bottom of the array, the smaller values rise (like bubbles) to the top. Now observe what the bubble sort does on its second pass through the array. The bubble sort begins the second pass by comparing the first value in the array with the second value. In this case, 8 is greater than 7, so the two values are interchanged as shown in the Second Comparison column in Figure 11-40. Notice that at this point, the data in the array is sorted.

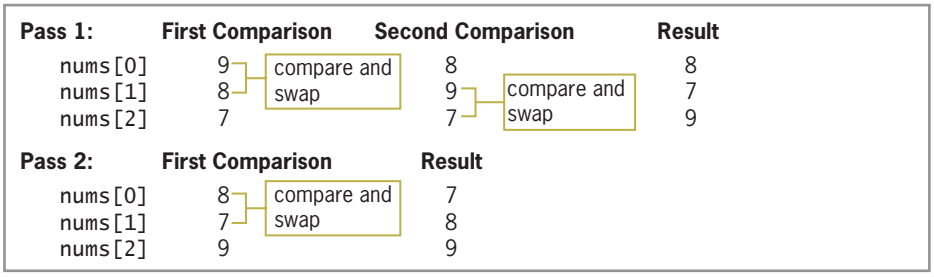


Figure 11-40 Array values before, during, and after the bubble sort

The program shown in Figure 11-41 uses the bubble sort to sort the contents of a four-element `int` array in ascending order. It then displays the contents of the sorted array on the screen.

```
 1  //Bubble Sort.cpp - uses the bubble sort to
 2  //sort the contents of a one-dimensional array
 3  //in ascending order
 4  //Created/revised by <your name> on <current date>
 5
 6  #include <iostream>
 7  using namespace std;
 8
 9  int main()
10  {
11     int numbers[4] = {23, 46, 12, 35};
12     int sub        = 0;   //keeps track of subscripts
13     int temp       = 0;   //variable used for swapping
14     int maxSub     = 3;   //maximum subscript
15     int lastSwap   = 0;   //position of last swap
16     char swap      = 'Y'; //indicates if a swap was made
17
18     //repeat loop instructions as long as a swap was made
19     while (swap == 'Y')
20     {
21          swap = 'N';    //assume no swaps are necessary
22
23          sub = 0;     //begin comparing with first
24                       //array element
25
26          //compare adjacent array elements to determine
27          //whether a swap is necessary
28          while (sub < maxSub)
29          {
30               if (numbers[sub] > numbers[sub + 1])
31               {
32                    //a swap is necessary
33                    temp = numbers[sub];
34                    numbers[sub] = numbers[sub + 1];
35                    numbers[sub + 1] = temp;
36                    swap = 'Y';
37                    lastSwap = sub;
38               } //end if
39               sub += 1; //increment subscript
40          } //end while
41
42          maxSub = lastSwap; //reset maximum subscript
43     } //end while
44
45     //display sorted array
46     for (int x = 0; x < 4; x += 1)
47          cout << numbers[x] << endl;
48     //end for
49
50     system("pause");
51     return 0;
52  }  //end of main function
```

your C++ development tool may not require this statement

Figure 11-41 Bubble sort program



If you already understand the bubble sort program's code, you can skip the remainder of this section and continue with the section titled *Parallel One-Dimensional Arrays*, which begins on Page 461.

To help you understand the bubble sort, you will desk-check the program shown in Figure 11-41. (However, refer to the TIP on this page.) The statements on Lines 11 through 16 create and initialize the `numbers` array and five variables. Figure 11-42 shows the desk-check table after these statements are processed.

numbers[0]	numbers[1]	numbers[2]	numbers[3]	
23	46	12	35	
sub	**temp**	**maxSub**	**lastSwap**	**swap**
0	0	3	0	Y

Figure 11-42 Desk-check table after the declaration statements on Lines 11 through 16 are processed

The condition in the `while` clause on Line 19 compares the contents of the `swap` variable with the letter Y. The condition evaluates to true; therefore, the computer processes the instructions in the body of the loop. The first two instructions appear on Lines 21 and 23. The instructions assign the letter N to the `swap` variable and assign the number 0 to the `sub` variable. The `while` clause on Line 28 begins a nested loop that repeats its instructions as long as the value stored in the `sub` variable is less than the value stored in the `maxSub` variable. At this point, the `sub` variable contains the number 0, and the `maxSub` variable contains the number 3; therefore, the computer processes the instructions in the nested loop. The condition in the `if (numbers[sub] > numbers[sub + 1])` clause on Line 30 determines whether the value stored in the `numbers[0]` variable is greater than the value stored in the `numbers[1]` variable. The condition evaluates to false, because the `numbers[0]` variable contains the number 23 and the `numbers[1]` variable contains the number 46. As a result, the instructions in the `if` statement's true path are skipped over and processing continues with the `sub += 1;` statement on Line 39. The statement adds the number 1 to the contents of the `sub` variable, giving 1. Figure 11-43 shows the desk-check table after the nested loop instructions are processed the first time. The new values entered in the table are shaded in the figure.

numbers[0]	numbers[1]	numbers[2]	numbers[3]	
23	46	12	35	
sub	**temp**	**maxSub**	**lastSwap**	**swap**
~~0~~	0	3	0	~~Y~~
~~0~~				N
1				

Figure 11-43 Desk-check table after the nested loop is processed the first time

Next, the computer evaluates the condition in the `while (sub < maxSub)` clause on Line 28. The condition evaluates to true, because the `sub` variable's value (1) is less than the `maxSub` variable's value (3). As a result,

the nested loop instructions are processed again. The condition in the `if (numbers[sub] > numbers[sub + 1])` clause on Line 30 determines whether the value stored in the `numbers[1]` variable is greater than the value stored in the `numbers[2]` variable. In this case, the condition evaluates to true, because the `numbers[1]` variable contains the number 46 and the `numbers[2]` variable contains the number 12. Because of this, the instructions in the `if` statement's true path are processed; the instructions appear on Lines 33 through 37. The first three instructions in the true path swap the values stored in the `numbers[1]` and `numbers[2]` variables. The fourth instruction, `swap = 'Y';`, assigns the letter Y to the `swap` variable to indicate that a swap was made. The last instruction in the true path, `lastSwap = sub;`, assigns the value stored in the `sub` variable—in this case, the number 1—to the `lastSwap` variable, which keeps track of the position of the last swap in the array. Next, the `sub += 1;` statement on Line 39 adds the number 1 to the contents of the `sub` variable; the result is 2. Figure 11-44 shows the desk-check table after the nested loop instructions are processed the second time. The new values entered in the table are shaded in the figure.

numbers[0]	numbers[1]	numbers[2]	numbers[3]	
23	~~46~~	~~12~~	35	
	12	46		

sub	temp	maxSub	lastSwap	swap
~~0~~	~~0~~	3	~~0~~	~~Y~~
~~0~~	46		1	~~N~~
~~1~~				Y
2				

Figure 11-44 Desk-check table after the nested loop is processed the second time

Next, the computer evaluates the condition in the `while (sub < maxSub)` clause on Line 28. The condition evaluates to true, because the `sub` variable's value (2) is less than the `maxSub` variable's value (3). Therefore, the computer once again processes the instructions in the nested loop. The condition in the `if (numbers[sub] > numbers[sub + 1])` clause on Line 30 determines whether the value stored in the `numbers[2]` variable is greater than the value stored in the `numbers[3]` variable. The condition evaluates to true, because the `numbers[2]` variable contains the number 46 and the `numbers[3]` variable contains the number 35. As a result, the computer processes the instructions in the `if` statement's true path. The first three instructions in the true path swap the values stored in the `numbers[2]` and `numbers[3]` variables. The fourth instruction in the true path assigns the letter Y to the `swap` variable to indicate that a swap was made. The last instruction in the true path assigns the value stored in the `sub` variable (2) to the `lastSwap` variable. Next, the `sub += 1;` statement on Line 39 adds the number 1 to the contents of the `sub` variable; the result is 3. Figure 11-45 shows the desk-check table after the nested loop instructions are processed the third time. The new values entered in the table are shaded in the figure.

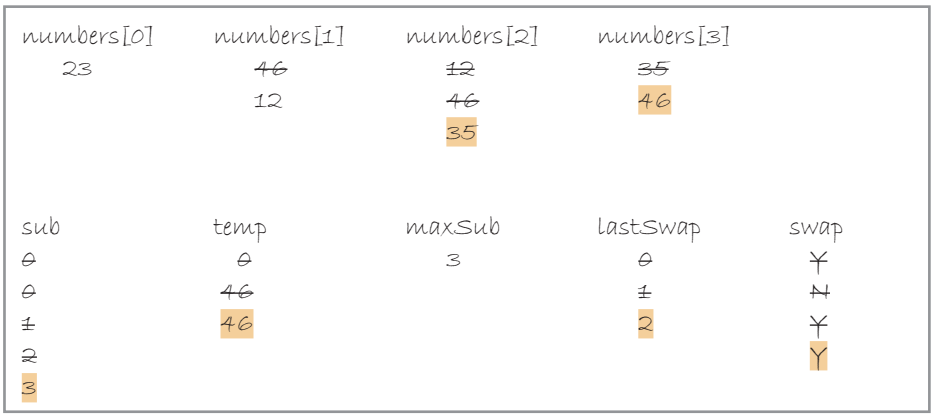


Figure 11-45 Desk-check table after the nested loop is processed the third time

The computer evaluates the condition in the `while (sub < maxSub)` clause on Line 28 next. The condition evaluates to false, because the `sub` variable's value (3) is not less than the `maxSub` variable's value (3). As a result, the nested loop instructions are skipped over and processing continues with the `maxSub = lastSwap;` statement on Line 42. The statement assigns the number 2 to the `maxSub` variable. Figure 11-46 shows the desk-check table after the outer loop instructions are processed the first time. The new value entered in the table is shaded in the figure.

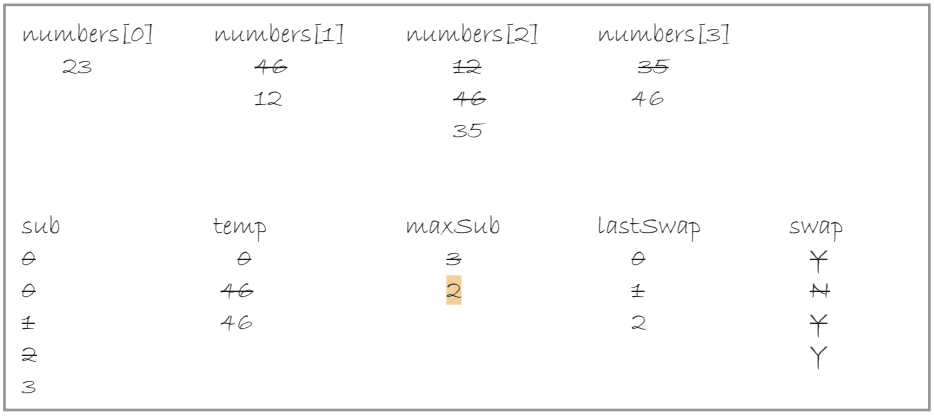


Figure 11-46 Desk-check table after the outer loop is processed the first time

The condition in the `while (swap == 'Y')` clause on Line 19 is processed next. The condition evaluates to true, so the computer processes the outer loop's instructions again. The first two instructions in the outer loop assign the letter N to the `swap` variable and assign the number 0 to the `sub` variable, as shown in Figure 11-47. The new values entered in the table are shaded in the figure.

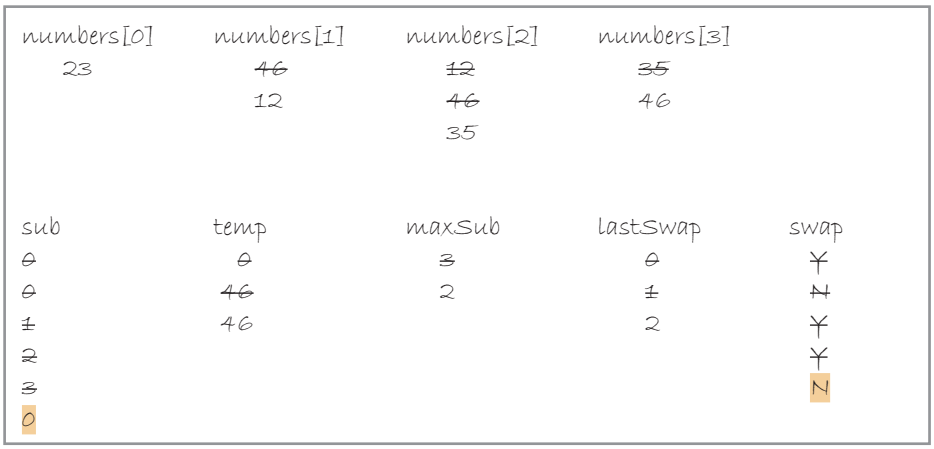


Figure 11-47 Desk-check table after the instructions on Lines 21 and 23 are processed

Next, the computer evaluates the condition in the `while (sub < maxSub)` clause on Line 28. The condition evaluates to true, so the computer processes the instructions in the nested loop. The condition in the `if (numbers[sub] > numbers[sub + 1])` clause on Line 30 determines whether the value stored in the `numbers[0]` variable is greater than the value stored in the `numbers[1]` variable. The condition evaluates to true, because the `numbers[0]` variable contains the number 23 and the `numbers[1]` variable contains the number 12. As a result, the computer processes the instructions in the `if` statement's true path. The first three instructions in the true path swap the values stored in the `numbers[0]` and `numbers[1]` variables. The fourth instruction in the true path assigns the letter Y to the `swap` variable to indicate that a swap was made. The last instruction in the true path assigns the value stored in the `sub` variable (0) to the `lastSwap` variable. Next, the `sub += 1;` statement on Line 39 adds the number 1 to the contents of the `sub` variable, giving 1. Figure 11-48 shows the desk-check table after the instructions in the nested loop are processed. The new values entered in the table are shaded in the figure.

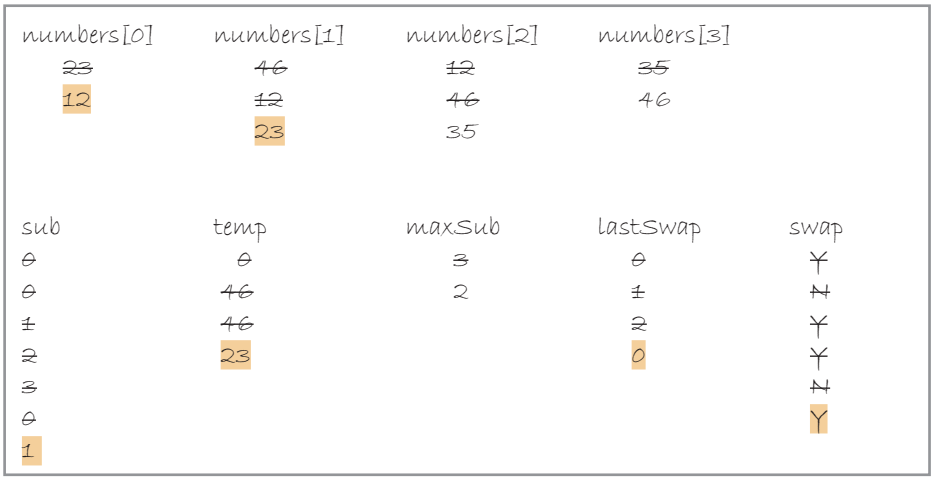


Figure 11-48 Desk-check table after the instructions in the nested loop are processed

The computer evaluates the condition in the `while (sub < maxSub)` clause on Line 28 next. The condition evaluates to true, because the `sub` variable's value (1) is less than the `maxSub` variable's value (2). Therefore, the computer processes the nested loop instructions once again. The condition in the `if (numbers[sub] > numbers[sub + 1])` clause on Line 30 determines whether the value stored in the `numbers[1]` variable is greater than the value stored in the `numbers[2]` variable. The condition evaluates to false, because the `numbers[1]` variable contains the number 23 and the `numbers[2]` variable contains the number 35. As a result, the computer skips over the instructions in the `if` statement's true path. Processing continues with the `sub += 1;` statement on Line 39. The statement increments the `sub` variable's value by 1; the result is 2. The condition in the `while (sub < maxSub)` clause on Line 28 is processed next. The condition evaluates to false, because the `sub` variable's value (2) is not less than the `maxSub` variable's value (2). Because of this, the computer skips over the instructions in the nested loop. Processing continues with the `maxSub = lastSwap;` statement on Line 42. The statement assigns the number 0 to the `maxSub` variable. Figure 11-49 shows the current status of the desk-check table. The new values entered in the table are shaded in the figure.

numbers[0]	numbers[1]	numbers[2]	numbers[3]
~~23~~	~~46~~	~~12~~	~~35~~
12	~~12~~	~~46~~	46
	23	35	

sub	temp	maxSub	lastSwap	swap
~~0~~	~~0~~	~~3~~	~~0~~	~~Y~~
~~0~~	~~46~~	~~2~~	~~1~~	~~N~~
~~1~~	~~46~~	0	~~2~~	~~Y~~
~~2~~	23		0	~~Y~~
~~3~~				~~N~~
~~0~~				Y
~~1~~				
2				

Figure 11-49 Desk-check table after the instructions in the nested loop are processed again

The computer evaluates the condition in the `while (swap == 'Y')` clause on Line 19 next. The condition evaluates to true, so the computer processes the outer loop's instructions again. The first two instructions in the outer loop assign the letter N to the `swap` variable and assign the number 0 to the `sub` variable. Next, the computer evaluates the condition in the `while (sub < maxSub)` clause on Line 28. The condition evaluates to false, because the `sub` variable's value (0) is not less than the `maxSub` variable's value (0). As a result, the computer skips over the instructions in the nested loop and continues processing with the `maxSub = lastSwap;` statement on Line 42. The statement assigns the number 0 to the `maxSub` variable. Figure 11-50 shows the current status of the desk-check table. The new values entered in the table are shaded in the figure.

numbers[0]	numbers[1]	numbers[2]	numbers[3]
~~23~~	~~46~~	~~12~~	~~35~~
12	~~12~~	~~46~~	46
	23	35	

sub	temp	maxSub	lastSwap	swap
~~0~~	~~0~~	~~3~~	~~0~~	~~Y~~
~~0~~	~~46~~	~~2~~	~~1~~	~~N~~
~~1~~	~~46~~	~~0~~	~~2~~	~~Y~~
~~2~~	23	0	0	~~Y~~
~~3~~				~~N~~
~~0~~				~~Y~~
~~1~~				N
~~2~~				
0				

Figure 11-50 Current status of the desk-check table

The condition in the `while (swap == 'Y')` clause on Line 19 is processed next and evaluates to false. Because of this, the computer skips over the instructions in the outer loop. Processing continues with the `for` clause on Line 46. The clause tells the computer to repeat the `cout << numbers[x] << endl;` statement four times: once for each element in the array. Figure 11-51 shows the result of running the bubble sort program.

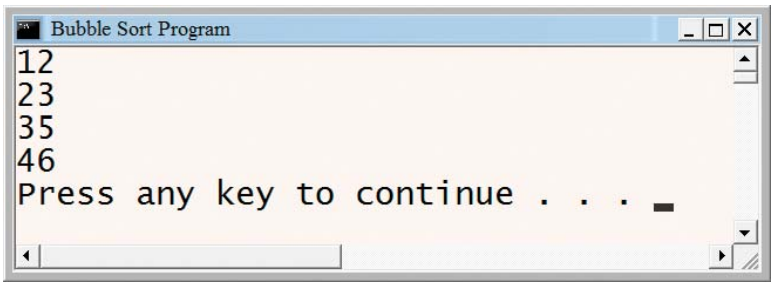


Figure 11-51 Result of running the bubble sort program

Parallel One-Dimensional Arrays

Figure 11-52 shows the problem specification, IPO chart information, and C++ instructions for the motorcycle club membership program. The program displays the annual fee associated with the membership type entered by the user. Notice that the program uses two one-dimensional arrays: a `char` array named `types` and an `int` array named `fees`. The `types` array stores the five membership types, and the `fees` array stores the annual fees associated with those types. Notice that the first element in each array pertains to membership type A; the `types` array contains the letter A, and the `fees` array contains the corresponding fee (100). The second element in each array pertains to membership type B, and so on. The two arrays are referred to as **parallel arrays**, because their elements are related by their position

(subscript) in the arrays. In other words, each element in the `types` array corresponds to the element located in the same position in the `fees` array. To determine the annual fee, you simply locate the membership type in the `types` array and then view its corresponding element in the `fees` array.

Problem specification
The members of a local motorcycle club are required to pay an annual fee based on their membership type. Create a program that displays a member's annual fee, as well as his or her membership type. The membership types and associated fees are shown here. Use a one-dimensional `char` array named `types` to store the membership types. Use a one-dimensional `int` array named `fees` to store the annual fees.

Membership type	Annual fee
A	100
B	110
C	125
D	150
E	200

IPO chart information	C++ instructions
Input	
membership type (A, B, C, D, or E)	`char memberType = ' ';`
Processing	
array (5 elements to store the types)	`char types[5] = {'A', 'B', 'C', 'D', 'E'};`
array (5 elements to store the fees)	`int fees[5] = {100, 110, 125, 150, 200};`
subscript counter (0 to 4)	`int sub = 0;`
Output	
fee	from the fees array
membership type	from the types array
Algorithm	

1. enter the membership type	`cout << "Membership type (A, B, C, D, or E): ";`
	`cin >> memberType;`
	`memberType = toupper(memberType);`

2. repeat while (the subscript counter is less than 5 and the membership type has not been located in the types array)	`while (sub < 5 && types[sub] != memberType)`
add 1 to the subscript counter	`sub += 1;`
end repeat	`//end while`

Figure 11-52 Problem specification, IPO chart information, and C++ instructions for the club membership program *(continues)*

(continued)

```
3. if (the subscript counter is less than 5)      if (sub < 5)
      display the membership type from the           cout << "Annual fee for
      types array and the fee from the               membership type " << types[sub]
      fees array, use the subscript counter          << ": $" << fees[sub] << endl;
      as the subscript
   else                                           else
      display "Invalid membership type"              cout << "Invalid membership
                                                     type" << endl;
   end if                                         //end if
```

Figure 11-52 Problem specification, IPO chart information, and C++ instructions for the club membership program

Figure 11-53 shows the code for the entire club membership program. The program declares and initializes the two parallel arrays (`types` and `fees`). It also declares and initializes two variables named `memberType` and `sub`. The `memberType` variable will store the membership type entered by the user, and the `sub` variable will keep track of the array subscripts. The program prompts the user to enter a membership type, and it stores the user's response in the `memberType` variable. The program then converts the contents of the `memberType` variable to uppercase. The `while` loop on Lines 24 through 26 is processed next. The loop will continue to increment the `sub` variable's value by 1 as long as the variable contains a value that is less than 5 and (at the same time) the membership type has not been located in the `types` array. The loop will stop when either of the following conditions is true: the `sub` variable contains the number 5 (which indicates that the loop reached the end of the array without finding the membership type) or the membership type is located in the array. After the loop completes its processing, the `if` statement in the program compares the number stored in the `sub` variable with the number 5. If the `sub` variable contains a number that is less than 5, it indicates that the loop stopped processing because the membership type was located in the `types` array. In that case, the `cout` statement on Lines 32 and 33 displays both the membership type from the `types` array and the corresponding annual fee from the `fees` array. However, if the `sub` variable contains a number that is not less than 5, it indicates that the loop stopped processing because it reached the end of the `types` array without finding the membership type. In that case, the `cout` statement on Line 35 displays the message "Invalid membership type". Figure 11-54 shows a sample run of the program.

```
 1  //Club Membership.cpp - displays the annual
 2  //membership fee
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  using namespace std;
 7
 8  int main()
 9  {
10      //declare arrays
11      char types[5]   = {'A', 'B', 'C', 'D', 'E'};
12      int fees[5]     = {100, 110, 125, 150, 200};
13      //declare variables
14      char memberType = ' ';
15      int sub         = 0;
16
17      //get type to search for
18      cout << "Membership type (A, B, C, D, or E): ";
19      cin >> memberType;
20      memberType = toupper(memberType);
21
22      //locate the position of the membership
23      //type in the types array
24      while (sub < 5 && types[sub] != memberType)
25            sub += 1;
26      //end while
27
28      //if the membership type was located in the
29      //types array, display the membership type
30      //and the corresponding fee
31      if (sub < 5)
32            cout << "Annual fee for membership type "
33            << types[sub] << ": $" << fees[sub] << endl;
34      else
35            cout << "Invalid membership type" << endl;
36      //end if
37
38      system("pause");
39      return 0;
40  }   //end of main function
```

parallel one-dimensional arrays

your C++ development tool may not require this statement

Figure 11-53 Club membership program

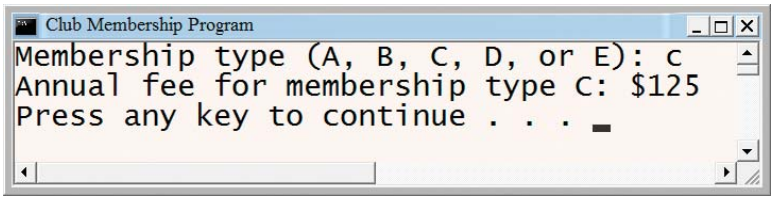


Figure 11-54 Sample run of the cub membership program

Mini-Quiz 11-3



The answers to Mini-Quiz questions are located in Appendix A.

1. Write a C++ `if` clause that determines whether the value stored in the `prices[x]` variable is less than the value stored in the `lowest` variable. The array and variable have the `double` data type.

2. The process of arranging data in alphabetical or numerical order is called _____.

3. Write a `for` loop that subtracts the number 3 from each of the 10 elements in an `int` array named `orders`. Use a variable named `x` to keep track of the array subscripts. Initialize the `x` variable to 0.



LAB 11-1 Stop and Analyze



The answers to the labs are located in Appendix A.

Study the program shown in Figure 11-55, and then answer the questions. The `domestic` array contains the amounts the company sold domestically during the months of January through June. The `international` array contains the amounts the company sold internationally during the same period.

```
1  //Lab11-1.cpp - calculates the total sales
2  //Created/revised by <your name> on <current date>
3
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9      //declare arrays and variable
10     int domestic[6] = {12000, 45000, 32000,
11                        67000, 24000, 55000};
12     int international[6] = {10000, 56000, 42000,
13                             23000, 12000, 34000};
14     int totalSales = 0;     //accumulator
15
16     //accumulate sales
17     for (int x = 0; x < 6; x += 1)
18          totalSales += domestic[x] + international[x];
19     //end for
20
21     //display total sales
22     cout << "Total sales: $" << totalSales << endl;
23
24     system("pause");
25     return 0;
26 }   //end of main function
```

Figure 11-55 Code for Lab 11-1

QUESTIONS

1. What relationship exists between the `domestic` and `international` arrays?
2. What value is stored in the `domestic[1]` element?
3. How can you calculate the total company sales made in February?
4. What is the highest subscript in the `international` array?
5. If you change the `for` clause in Line 17 to `for (int x = 1; x <= 6; x += 1)`, how will the change affect the assignment statement in the `for` loop?
6. Follow the instructions for starting C++ and opening the Lab11-1.cpp file. The file is contained in either the Cpp6\Chap11\Lab11-1 Project folder or the Cpp6\Chap11 folder. Run the program. The total company sales are $412000.
7. Modify the program so that it displays the total domestic sales, total international sales, and total company sales. Save and then run the program.
8. Now modify the program so that it also displays the total sales made in each month. Use month numbers from 1 through 6. Save and then run the program.



LAB 11-2 Plan and Create

In this lab, you will plan and create an algorithm for Penelope Havert. The problem specification, IPO chart information, and C++ instructions are shown in Figure 11-56. According to the figure, the `main` function first will get the 12 rainfall amounts from the user. It then will display a menu that allows the user to select from three different options. If the user chooses to display the monthly rainfall amounts, the `main` function will call the `displayMonthly` function, passing it the `rainfall` array and the number of array elements. The `displayMonthly` function uses a `for` loop to display the 12 monthly rainfall amounts on the screen. However, if the user chooses to display the total rainfall amount, the `main` function will call the `displayTotal` function, passing it the `rainfall` array and the number of array elements. The `displayTotal` function uses a `for` loop to accumulate the 12 monthly rainfall amounts. It then displays the total rainfall amount on the screen. The program ends when the user enters any number other than 1 or 2 in response to the "Enter your choice:" prompt.

Problem specification
Penelope Havert wants a program that allows her to enter the monthly rainfall amounts for the previous year. The program then should allow her to either display the monthly rainfall amounts on the screen or calculate and display the total annual rainfall amount. In this program, you will use two program-defined void functions named `displayMonthly` and `displayTotal`. Both functions will be passed the contents of a one-dimensional array named `rainfall`, along with the number of array elements. The `rainfall` array will contain the 12 monthly rainfall amounts entered by the user. Void functions are appropriate in this case because neither function needs to return a value to the statement that called it.

main function

IPO chart information	C++ instructions
Input	
menu choice	`int choice = 0;`
monthly rainfall amount (12)	`double rainfall[12] = {0.0};`
Processing	
subscript counter (0 to 11)	declared and initialized in the for clause
Output	
array (12 element)	displayed by the displayMonthly function
total rainfall	displayed by the displayTotal function
Algorithm	
1. repeat for (subscript counter from 0 to 11)	`for (int x = 0; x < 12; x += 1)`
	`{`
enter monthly rainfall amount	`cout << "Enter rainfall for month " << x + 1 << ": ";`
	`cin >> rainfall[x];`
end repeat	`} //end for`
2. repeat	`do //begin loop`
	`{`
display menu	`cout << endl;`
	`cout << "1 Display monthly amounts" << endl;`
	`cout << "2 Display total amount" << endl;`
	`cout << "3 End program" << endl;`
enter the menu choice	`cout << "Enter your choice: ";`
	`cin >> choice;`
if (the menu choice is 1)	`if (choice == 1)`
call the displayMonthly function to display the array, pass the array and the	`displayMonthly(rainfall, 12);`

Figure 11-56 Problem specification, IPO chart information, and C++ instructions for Lab 11-2 *(continues)*

(continued)

```
        number of elements
    else                                        else
        if (the menu choice is 2)                  if (choice == 2)
            call the displayTotal                     displayTotal(rainfall, 12);
            function to calculate
            and display the total
            rainfall, pass the array
            and the number of
            elements
        end if                                     //end if
    end if                                      //end if
while (menu choice equals 1 or 2)            }  while (choice == 1 ||
                                             choice == 2);
```

displayMonthly function

IPO chart information	C++ instructions
Input	
array	`double rain[]` (formal parameter)
number of elements	`int numElements` (formal parameter)
Processing	
subscript counter (0 to 11)	declared and initialized in the for clause
Output	
array element	from array
Algorithm	
1. display heading	`cout << "Monthly rainfall amounts:" << endl;`
2. repeat for (subscript counter from 0 to 11)	`for (int x = 0; x < numElements; x += 1)`
display the current array element	`cout << rain[x] << endl;`
end repeat	`//end for`

displayTotal function

IPO chart information	C++ instructions
Input	
array	`double rain[]` (formal parameter)
number of elements	`int numElements` (formal parameter)
Processing	
subscript counter (0 to 11)	declared and initialized in the for clause
Output	
total rainfall	`double total = 0.0;`
Algorithm	
1. repeat for (subscript counter from 0 to 11)	`for (int x = 0; x < numElements; x += 1)`
add the current array element to the total rainfall	`total = total + rain[x];`
end repeat	`//end for`
2. display the total rainfall	`cout << "Total rainfall: " << total << endl;`

Figure 11-56 Problem specification, IPO chart information, and C++ instructions for Lab 11-2

Figure 11-57 shows the code for the entire rainfall program, and Figure 11-58 shows the completed desk-check table for the program, assuming the user enters the following 12 rainfall amounts: 2.44, 2.36, 2.76, 1.2, .4, .07, .04, .23, .54, .63, 1.54, and 2.16. After entering the rainfall amounts, the user selects choice 1, followed by choice 2, followed by choice 3.

```
 1  //Lab11-2.cpp
 2  //Stores monthly rainfall amounts in an array
 3  //Displays the monthly rainfall amounts or the
 4  //total annual rainfall amount
 5  //Created/revised by <your name> on <current date>
 6
 7  #include <iostream>
 8  using namespace std;
 9
10  //function prototypes
11  void displayMonthly(double rain[], int numElements);
12  void displayTotal(double rain[], int numElements);
13
14  int main()
15  {
16      //declare array and variable
17      double rainfall[12] = {0.0};
18      int choice          = 0;
19
20      //get rainfall amounts
21      for (int x = 0; x < 12; x += 1)
22      {
23            cout << "Enter rainfall for month "
24            << x + 1 << ": ";
25            cin >> rainfall[x];
26      }     //end for
27
28      do
29      {
30            //display menu and get menu choice
31            cout << endl;
32            cout << "1 Display monthly amounts" << endl;
33            cout << "2 Display total amount" << endl;
34            cout << "3 End program" << endl;
35            cout << "Enter your choice: ";
36            cin >> choice;
37
38            //call appropriate function or end program
39            if (choice == 1)
40                  displayMonthly(rainfall, 12);
41            else
42                  if (choice == 2)
43                        displayTotal(rainfall, 12);
44                  //end if
45            //end if
46      } while (choice == 1 || choice == 2);
```

Figure 11-57 Rainfall program *(continues)*

(continued)

```
47
48      system("pause");
49     return 0;
50  }  //end of main function
51
52  //*****function definitions*****
53  void displayMonthly(double rain[], int numElements)
54  {
55      cout << "Monthly rainfall amounts:" << endl;
56      for (int x = 0; x < numElements; x += 1)
57            cout << rain[x] << endl;
58      //end for
59  }  //end of displayMonthly function
60
61  void displayTotal(double rain[], int numElements)
62  {
63      double total = 0.0;
64      for (int x = 0; x < numElements; x += 1)
65            total = total + rain[x];
66      //end for
67      cout << "Total rainfall: " << total << endl;
68  }  //end of displayTotal function
```

your C++ development tool may not require this statement

Figure 11-57 Rainfall program

~~rain[0]~~	~~rain[1]~~	~~rain[2]~~	~~rain[3]~~
~~rain[0]~~	~~rain[1]~~	~~rain[2]~~	~~rain[3]~~
rainfall[0]	rainfall[1]	rainfall[2]	rainfall[3]
~~0.0~~	~~0.0~~	~~0.0~~	~~0.0~~
2.44	2.36	2.76	1.2
~~rain[4]~~	~~rain[5]~~	~~rain[6]~~	~~[rain7]~~
~~rain[4]~~	~~rain[5]~~	~~rain[6]~~	~~[rain7]~~
rainfall[4]	rainfall[5]	rainfall[6]	rainfall[7]
~~0.0~~	~~0.0~~	~~0.0~~	~~0.0~~
.4	.07	.04	.23
rain[8]	rain[9]	rain[10]	rain[11]
rain[8]	rain[9]	rain[10]	rain[11]
rainfall[8]	rainfall[9]	rainfall[10]	rainfall[11]
~~0.0~~	~~0.0~~	~~0.0~~	~~0.0~~
.54	.63	1.54	2.16

Figure 11-58 Completed desk-check table for the rainfall program *(continues)*

(continued)

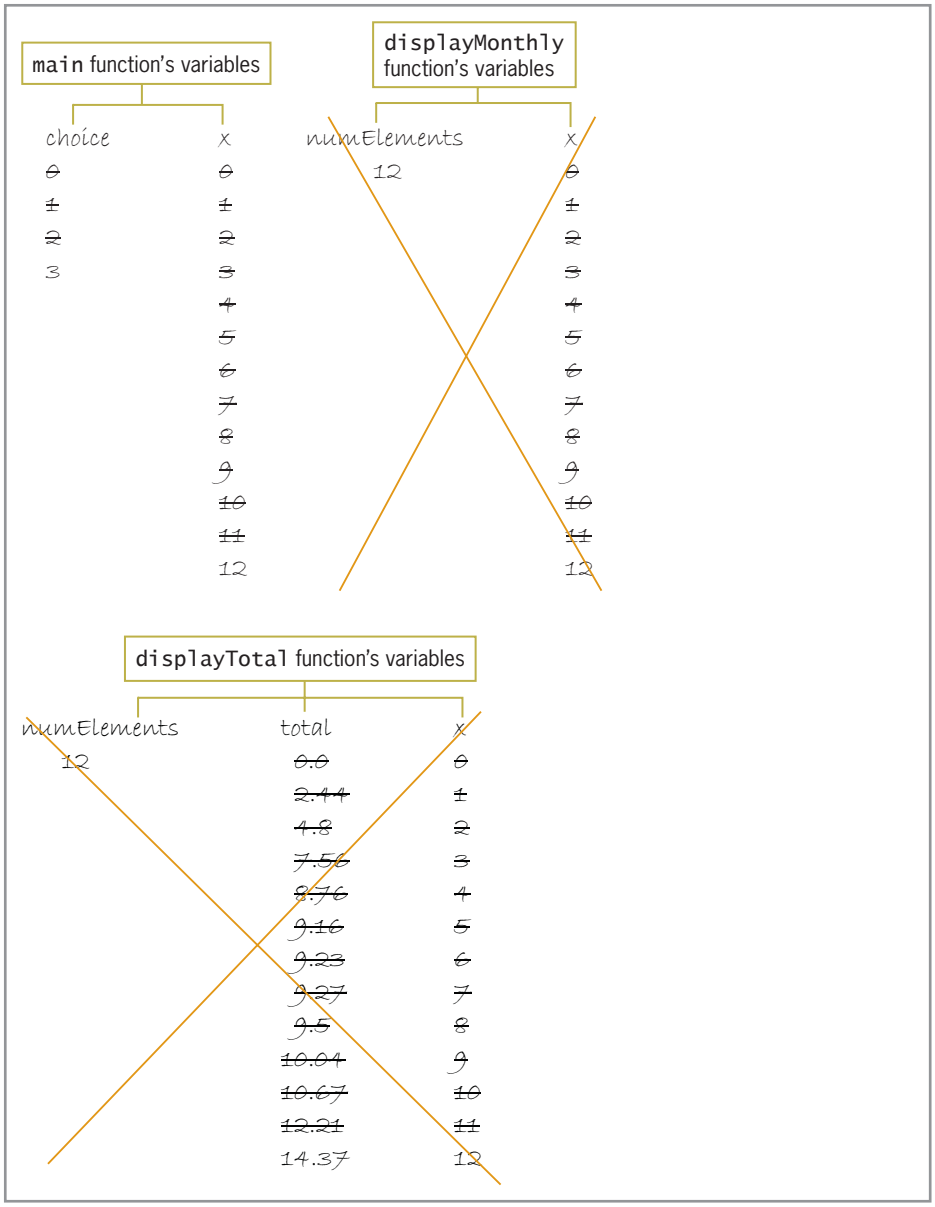


Figure 11-58 Completed desk-check table for the rainfall program

The final step in the problem-solving process is to evaluate and modify (if necessary) the program. Recall that you evaluate a program by entering its instructions into the computer and then using the computer to run (execute) it. While the program is running, you enter the same sample data used when desk-checking the program.

DIRECTIONS

Follow the instructions for starting your C++ development tool. Depending on the development tool you are using, you may need to create a new project; if so, name the project Lab11-2 Project and save it in the Cpp6\Chap11 folder. Enter the instructions shown in Figure 11-57 in a source file named Lab11-2.cpp. (Do not enter the line numbers.) Save the file in either the project folder or the Cpp6\Chap11 folder. Now follow the appropriate

instructions for running the Lab11-2.cpp file. Test the program using the same data you used to desk-check the program. (The total rainfall amount should be 14.37.) If necessary, correct any bugs (errors) in the program.



LAB 11-3 Modify

If necessary, create a new project named Lab11-3 Project. Enter (or copy) the Lab11-2.cpp instructions into a new source file named Lab11-3.cpp. Change Lab11-2.cpp in the first comment to Lab11-3.cpp. Make the following two modifications to the program. First, change the void `displayTotal` function to a value-returning function named `getTotal`. The `getTotal` function should calculate the total rainfall and then return the result to the `main` function for displaying on the screen. Second, the `main` function should display an appropriate message when the user enters a menu choice other than 1, 2, or 3. It then should display the menu again. Save and then run the program. Test the program appropriately.



LAB 11-4 Desk-Check

Desk-check the code in Figure 11-59 using the data shown below. What will the `for` loop on Lines 31 through 34 display on the screen?

Student	Midterm	Final
1	90	100
2	88	68
3	77	75
4	85	85
5	45	32

```
 1  //Lab11-4.cpp
 2  //Stores averages in a one-dimensional array
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  using namespace std;
 7
 8  int main()
 9  {
10     //declare arrays
11     double midterms[5] = {0.0};
12     double finals[5]   = {0.0};
13     double averages[5] = {0.0};
14
```

Figure 11-59 Code for Lab 11-4 *(continues)*

(continued)

```
15     //get exam scores
16     for (int x = 0; x < 5; x += 1)
17     {
18          cout << "Midterm exam score for student "
19               << x + 1 << ": ";
20          cin >> midterms[x];
21          cout << "Final exam score for student "
22               << x + 1 << ": ";
23          cin >> finals[x];
24          cout << endl;
25          //calculate and assign average
26          averages[x] = (midterms[x] + finals[x]) / 2;
27   }  //end for
28
29     //display contents of array
30     cout << endl;
31     for (int y = 0; y < 5; y += 1)
32          cout << "Student " << y + 1 << " average: "
33               << averages[y] << endl;
34     //end for
35
36     system("pause");
37     return 0;
38   }  //end of main function
```

your C++ development tool may not require this statement

Figure 11-59 Code for Lab 11-4



LAB 11-5 Debug

Follow the instructions for starting C++ and opening the Lab11-5.cpp file. The file is contained in either the Cpp6\Chap11\Lab11-5 Project folder or the Cpp6\Chap11 folder. Debug the program.

Summary

- An array is a group of variables that have the same name and data type and are related in some way. The most commonly used arrays in programs are one-dimensional arrays and two-dimensional arrays.
- Programmers use arrays to temporarily store related data in the internal memory of the computer. By doing so, a programmer can increase the efficiency of a program, because data can be both stored in and retrieved from internal memory much faster than it can be written to and read from a file on a disk. In addition, after the data is entered into an array, the program can use the data as many times as it is needed.

- You must declare an array before you can use it. After declaring an array, you can use an assignment statement or the extraction operator to enter data into the array.
- Each of the array elements in a one-dimensional array is assigned a unique number, called a subscript. The first element is assigned a subscript of 0. The second element is assigned a subscript of 1, and so on. Because the first array subscript is 0, the last subscript in a one-dimensional array is always one number less than the number of elements.
- You refer to each element in a one-dimensional array by the array's name and the element's subscript, which is specified in square brackets immediately following the name.
- Parallel arrays are two or more arrays whose elements are related by their corresponding subscript (or position) in the arrays.

Key Terms

Array—a group of related variables that have the same name and data type

Bubble sort—one of many sorting algorithms used to sort small arrays; works by comparing adjacent array elements and swapping the ones that are out of order

Elements—the variables in an array

One-dimensional array—an array whose elements are identified by a unique subscript

Parallel arrays—two or more arrays whose elements are related by their corresponding position (subscript) in the arrays

Populating the array—refers to the process of initializing the elements in an array

Scalar variable—another term for a simple variable

Simple variable—a variable that is unrelated to any other variable in the computer's internal memory; also called a scalar variable

Sorting—the process of arranging data in a specific order

Subscript—a unique number that identifies the position of an element in an array

Review Questions

1. Which of the following is false?
   a. The elements in an array are related in some way.
   b. All of the elements in an array have the same data type.
   c. All of the elements in a one-dimensional array have the same subscript.
   d. The first element in a one-dimensional array has a subscript of 0 (zero).

2. Which of the following statements declares a five-element array named `population`?

   a. `int population[4] = {0};`

   b. `int population[5] = {0};`

   c. `int population[4] = 0`

   d. `int population[5] = {0}`

   Use the `sales` array to answer Review Questions 3 through 7. The array was declared using the `int sales[5] = {10000, 12000, 900, 500, 20000};` statement.

3. The `sales[3] = sales[3] + 10;` statement will replace the number _____.

   a. 500 with 10

   b. 500 with 510

   c. 900 with 910

   d. 900 with 910

4. The `sales[4] = sales[4 - 2];` statement will replace the number _____.

   a. 20000 with 900

   b. 20000 with 19998

   c. 500 with 12000

   d. 500 with 498

5. The `cout << sales[0] + sales[1] << endl;` statement will _____.

   a. display 22000

   b. display 10000 + 12000

   c. display `sales[0] + sales[1]`

   d. result in an error

6. Which of the following `if` clauses verifies that the array subscript stored in the `x` variable is valid for the `sales` array?

   a. `if (sales[x] >= 0 && sales[x] < 4)`

   b. `if (sales[x] >= 0 && sales[x] <= 4)`

   c. `if (x >= 0 && x < 4)`

   d. `if (x >= 0 && x <= 4)`

7. Which of the following will correctly add the number 100 to each variable in the `sales` array? The `x` variable was declared using the `int x = 0;` statement.

   a.
```
while (x <= 4)
      x += 100;
//end while
```

   b.
```
while (x <= 4)
{
      sales = sales + 100;
      x += 1;
}     //end while
```

   c.
```
while (sales < 5)
{
      sales[x] += 100;
}     //end while
```

   d.
```
while (x <= 4)
{
      sales[x] += 100;
      x += 1;
}     //end while
```

Use the `nums` array to answer Review Questions 8 through 12. The array was declared using the `int nums[4] = {10, 5, 7, 2};` statement. The `x` and `total` variables are `int` variables and are initialized to 0. The `avg` variable is a `double` variable and is initialized to 0.0.

8. Which of the following will correctly calculate the average of the elements included in the `nums` array?

   a.
```
while (x < 4)
{
      nums[x] = total + total;
      x += 1;
}     //end while
avg = static_cast<double>(total) /
static_cast<double>(x);
```

   b.
```
while (x < 4)
{
      total += nums[x];
      x += 1;
}     //end while
avg = static_cast<double>(total) /
static_cast<double>(x);
```

   c.
```
while (x < 4)
{
      total += nums[x];
      x += 1;
}     //end while
avg = static_cast<double>(total) /
static_cast<double>(x) - 1;
```

d.
```
while (x < 4)
{
     total += nums[x];
     x += 1;
}    //end while
avg = static_cast<double>(total) /
static_cast<double>(x - 1);
```

9. The code in Review Question 8's answer a assigns _____ to the `avg` variable.

   a. 0.0

   b. 5.0

   c. 6.0

   d. 8.0

10. The code in Review Question 8's answer b assigns _____ to the `avg` variable.

    a. 0.0

    b. 5.0

    c. 6.0

    d. 8.0

11. The code in Review Question 8's answer c assigns _____ to the `avg` variable.

    a. 0.0

    b. 5.0

    c. 6.0

    d. 8.0

12. The code in Review Question 8's answer d assigns _____ to the `avg` variable.

    a. 0.0

    b. 5.0

    c. 6.0

    d. 8.0

13. If the `cities` and `zips` arrays are parallel arrays, which of the following statements will display the city name associated with the zip code stored in the `zips[8]` variable?

    a. `cout << cities[zips[8]] << endl;`

    b. `cout << cities(zips[8]) << endl;`

    c. `cout << cities[8] << endl;`

    d. `cout << cities(8) << endl;`

Exercises



Pencil and Paper

TRY THIS

1. Write the statement to declare and initialize a one-dimensional `int` array named `numbers` that has 20 elements. Then write the statement to store the number 7 in the second element in the array. (The answers to TRY THIS Exercises are located at the end of the chapter.)

TRY THIS

2. Write the code to display the contents of the `numbers` array from Pencil and Paper Exercise 1. Use the `for` statement with a counter variable named `x`. (The answers to TRY THIS Exercises are located at the end of the chapter.)

MODIFY THIS

3. Rewrite the code from Pencil and Paper Exercise 2 using the `while` statement.

INTRODUCTORY

4. Write the statement to declare and initialize a one-dimensional `double` array named `rates` that has five elements. Use the following numbers to initialize the array: 6.5, 8.3, 4.0, 2.0, and 10.5.

INTRODUCTORY

5. Write the code to display the contents of the `rates` array from Pencil and Paper Exercise 4. Use the `for` statement.

INTRODUCTORY

6. Rewrite the code from Pencil and Paper Exercise 5 using the `do while` statement.

INTERMEDIATE

7. Write the statement to assign the C++ keyword `true` to the variable located in the third element in a one-dimensional `bool` array named `answers`.

INTERMEDIATE

8. Write the code to multiply by 2 the number stored in the first element in a one-dimensional `int` array named `nums`. Store the result in the `numDoubled` variable.

INTERMEDIATE

9. Write the code to add together the numbers stored in the first and second elements in a one-dimensional `int` array named `nums`. Display the sum on the screen.

INTERMEDIATE

10. Write the code to subtract the number 1 from each element in a one-dimensional `int` array named `quantities`. The array has 10 elements. Use the `while` statement.

INTERMEDIATE

11. Rewrite the code from Pencil and Paper Exercise 10 using the `for` statement.

INTERMEDIATE

12. Write the code to find the square root of the number stored in the first element in a one-dimensional `double` array named `mathNumbers`. Display the result on the screen.

13. Write the code to display the largest number stored in a one-dimensional `int` array named `orders`. The array has five elements. Use the `while` statement. ADVANCED

14. Rewrite the code from Pencil and Paper Exercise 13 using the `for` statement. ADVANCED

15. The `numbers` array is a five-element one-dimensional `int` array. The following statement should display the result of raising the first array element to the second power: `cout << pow(nums[0], 2);`. Correct the statement. SWAT THE BUGS



Computer

16. If necessary, create a new project named TryThis16 Project. Enter the C++ instructions from Figure 11-19 (in the chapter) into a source file named TryThis16.cpp. Change the filename in the first comment to TryThis16.cpp. Save and then run the program. Test the program using the data shown in Figure 11-20 in the chapter. Change each of the `for` loops in the program to `while` loops. Save and then run the program. Test the program using the data shown in Figure 11-20 in the chapter. (The answers to TRY THIS Exercises are located at the end of the chapter.) TRY THIS

17. If necessary, create a new project named TryThis17 Project. Enter the C++ instructions from Figure 11-22 (in the chapter) into a source file named TryThis17.cpp. Change the filename in the first comment to TryThis17.cpp. Save and then run the program. Create a void function named `getAverage`. The `getAverage` function should calculate the average number of pounds of coffee used. Hint: Pass the `average` variable *by reference*. (The answers to TRY THIS Exercises are located at the end of the chapter.) TRY THIS

18. In this exercise, you modify the random numbers program from the chapter. If necessary, create a new project named ModifyThis18 Project. Enter the C++ instructions from Figure 11-32 into a source file named ModifyThis18.cpp. Change the filename in the first comment to ModifyThis18.cpp. Add another program-defined value-returning function to the program. Name the function `getLowest`. The `getLowest` function should determine the lowest integer in the array. Save and then run the program. Test the program appropriately. MODIFY THIS

19. In this exercise, you modify the hourly rate program from the chapter. If necessary, create a new project named ModifyThis19 Project. Enter the C++ instructions from Figure 11-28 into a new source file named ModifyThis19.cpp. Change the filename in the first comment to ModifyThis19.cpp. Save and then run the program. Test the program appropriately. Now modify the program to use pay codes of A, B, C, D, E, and F. Store the pay codes in a parallel array, and then modify the program's code appropriately. Save and then run the program. Test the program appropriately. MODIFY THIS

INTRODUCTORY

20. Follow the instructions for starting C++ and opening the Introductory20.cpp file. The file is contained in either the Cpp6\Chap11\Introductory20 Project folder or the Cpp6\Chap11 folder. The program should calculate the average of the values stored in the `rates` array. It then should display the average rate on the screen. Complete the program using the `for` statement. Save and then run the program.

INTRODUCTORY

21. Follow the instructions for starting C++ and opening the Introductory21.cpp file. The file is contained in either the Cpp6\Chap11\Introductory21 Project folder or the Cpp6\Chap11 folder. The program should display the contents of the `orders` array. Complete the program using the `while` statement. Save and then run the program.

INTERMEDIATE

22. Follow the instructions for starting C++ and opening the Intermediate22.cpp file. The file is contained in either the Cpp6\Chap11\Intermediate22 Project folder or the Cpp6\Chap11 folder. The program should display the contents of the two parallel arrays. Complete the program using the `do while` statement. Save and then run the program.

INTERMEDIATE

23. In this exercise, you modify the program from Lab 11-2 in the chapter. If necessary, create a new project named Intermediate23 Project. Copy the instructions from the Lab11-2.cpp file into a source file named Intermediate23.cpp. (Alternatively, you can enter the instructions from Figure 11-57 into the Intermediate23.cpp file.) Change the filename in the first comment to Intermediate23.cpp. Add three additional functions to the program: `displayAverage`, `displayHigh`, and `displayLow`. The functions should display the average rainfall amount, the highest rainfall amount, and the lowest rainfall amount. Save and then run the program. Test the program appropriately.

INTERMEDIATE

24. If necessary, create a new project named Intermediate24 Project. Also create a new source file named Intermediate24.cpp. Declare a 12-element `int` array named `days`. Assign the number of days in each month to the array, using 28 for February. Code the program so that it displays the number of days corresponding to the month number entered by the user. For example, when the user enters the number 7, the program should display the number 31. The program also should display an appropriate message when the user enters an invalid month number. Use a sentinel value to end the program. Save and then run the program. Test the program using the valid numbers 1 through 12. Also test it using an invalid number, such as 20.

INTERMEDIATE

25. Follow the instructions for starting C++ and opening the Intermediate25.cpp file. The file is contained in either the Cpp6\Chap11\Intermediate25 Project folder or the Cpp6\Chap11 folder. Code the program so that it asks the user for a percentage amount by which each price should be increased. The program then should increase each price in the array by that amount. For example, when the user enters the number 15, the program should increase each element's value by 15%. After increasing each price, the program should display the contents of the array. Save and then run the program. Increase each price by 5%.

INTERMEDIATE

26. In this exercise, you modify the program from Computer Exercise 25. If necessary, create a new project named Intermediate26 Project. Also create a new source file named Intermediate26.cpp. Copy the C++ instructions from the Intermediate25.cpp file into the Intermediate26.cpp file. Change the filename in the first comment to Intermediate26.cpp. Modify the program so that it also asks the user to enter a number from 1 through 10. When the user enters the number 1, the program should update only the first price in the array. When the user enters the number 2, the program should update only the second price in the array, and so on. Use a loop that stops prompting the user when he or she enters a number that is either less than or equal to 0 or greater than 10. Save and then run the program. Increase the second price by 10%. Next, increase the tenth price by 2%. Finally, decrease the first price by 10%.

ADVANCED

27. Follow the instructions for starting C++ and opening the Advanced27.cpp file. The file is contained in either the Cpp6\Chap11\Advanced27 Project folder or the Cpp6\Chap11 folder. Enter the code that prompts the user to enter a score from 0 through 100. The program should display the number of students earning that score. Use a sentinel value to end the program. Save and then run the program. Use the program to answer the following questions.

    How many students earned a score of 72?

    How many students earned a score of 88?

    How many students earned a score of 20?

    How many students earned a score of 99?

ADVANCED

28. In this exercise, you modify the program from Computer Exercise 27. If necessary, create a new project named Advanced28 Project. Also create a new source file named Advanced28.cpp. Copy the C++ instructions from the Advanced27.cpp file into the Advanced28.cpp file. Change the filename in the first comment to Advanced28.cpp. Modify the program so that it prompts the user to enter a minimum score and a maximum score. The program should display the number of students who earned a score within that range. Use a sentinel value to end the program. Save and then run the program. Use the program to answer the following questions.

    How many students earned a score from 70 through 79?

    How many students earned a score from 65 through 85?

    How many students earned a score from 0 through 50?

ADVANCED

29. In this exercise, you create a program that generates and displays six unique random integers for a lottery game. Each lottery number can range from 1 through 54 only. If necessary, create a new project named Advanced29 Project. Also create a new source file named Advanced29.cpp. Create a program that generates six unique random integers from 1 through 54, and then displays the integers on the screen. (Hint: Store the numbers in a one-dimensional array.) Save and then run the program.

ADVANCED

30. In this exercise, you create a program that uses two parallel one-dimensional arrays. Ms. Jenkins uses the grade table shown in Figure 11-60 for her Introduction to Programming course. She wants a program that displays the grade after she enters the total points earned. If necessary, create a new project named Advanced30 Project. Also create a new source file named Advanced30.cpp. Store the minimum points in a one-dimensional `int` array. Store the grades in a one-dimensional `char` array. Use a sentinel value to stop the program. Save and then run the program. Test the program using the following amounts: 455, 210, 400, and 349.

Minimum points	Maximum points	Grade
0	299	F
300	349	D
350	399	C
400	449	B
450	500	A

Figure 11-60

ADVANCED

31. In this exercise, you modify the program from Computer Exercise 30. The modified program allows the user to change the grading scale while the program is running. If necessary, create a new project named Advanced31 Project. Also create a new source file named Advanced31.cpp. Copy the instructions from the Advanced30.cpp file into the Advanced31.cpp file. Change the filename in the first comment. Modify the program so that it allows the user to enter the total number of possible points—in other words, the total number of points a student can earn in the course—when the program is run. Also modify the program so that it uses the grading scale shown in Figure 11-61. For example, when the user enters the number 500 as the total number of possible points, the program should use 450 (which is 90% of 500) as the minimum number of points for an A. When the user enters the number 300 as the total number of possible points, the program should use 270 (which is 90% of 300) as the minimum number of points for an A. Save and then run the program. Test the program using 300 as the total number of possible points and 185 as the number of points earned. The program should display D as the grade. Stop the program. Then test it using 500 and 363 as the total number of possible points and the total points earned, respectively. The program should display C as the grade.

Minimum points	Grade
0	F
60% of the possible points	D
70% of the possible points	C
80% of the possible points	B
90% of the possible points	A

Figure 11-61

ADVANCED

32. In this exercise, you create a program that uses two parallel one-dimensional arrays. The program displays a shipping charge that is based on the number of items ordered by a customer. The shipping charge scale is shown in Figure 11-62. If necessary, create a new project named Advanced32 Project. Also create a new source file named Advanced32.cpp. Store the maximum order amounts in a one-dimensional `int` array. Store the shipping charge amounts in a parallel one-dimensional `int` array. The program should allow the user to enter the number of items a customer ordered. It then should display the appropriate shipping charge. Use a sentinel value to stop the program. Save and then run the program. Test the program appropriately.

Minimum order	Maximum order	Shipping charge
1	10	15
11	50	10
51	100	5
101	99999	0

Figure 11-62

ADVANCED

33. In this exercise, you code a program that uses three parallel numeric arrays. The program searches one of the arrays and then displays the corresponding values from the other two arrays. Follow the instructions for starting C++ and opening the Advanced33.cpp file. The file is contained in either the Cpp6\Chap11\Advanced33 Project folder or the Cpp6\Chap11 folder. The program should prompt the user to enter a product ID. It then should search for the product ID in the `ids` array and display the corresponding price and quantity from the `prices` and `quantities` arrays. Allow the user to display the price and quantity for as many product IDs as desired without having to execute the program again. Save and then run the program. Test the program appropriately.

SWAT THE BUGS

34. Follow the instructions for starting C++ and opening the SwatTheBugs34.cpp file. The file is contained in either the Cpp6\Chap11\SwatTheBugs34 Project folder or the Cpp6\Chap11 folder. Debug the program.

Answers to TRY THIS Exercises



Pencil and Paper

1.
```
int numbers[20] = {0};
 numbers[1] = 7;
```

2.
```
for (int x = 0; x < 20; x += 1)
      cout << numbers[x] << endl;
 //end for
```



Computer

16. See Figure 11-63. The changes are shaded in the figure.

```
 1  //TryThis16.cpp - displays the contents
 2  //of an array
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  #include <iomanip>
 7  using namespace std;
 8
 9  //function prototype
10  void displayArray(double dollars[], int numElements);
11
12  int main()
13  {
14     //declare array
15     double sales[4] = {0.0};
16
17     //fill array with data
18     int sub = 0;
19     while (sub < 4)
20     {
21          cout << "Enter the sales for Region ";
22          cout << sub + 1 << ": ";
23          cin >> sales[sub];
24          sub += 1;
25     }    //end while
26
27     //display the contents of the array
28     displayArray(sales, 4);
29
30     system("pause");
31     return 0;
32  }  //end of main function
33
34  //*****function definitions*****
35  void displayArray(double dollars[], int numElements)
36  {
37     cout << fixed << setprecision(2) << endl << endl;
38     int sub = 0;
39     while (sub < numElements)
40     {
41          cout << "Sales for Region " << sub + 1 << ": $";
42          cout << dollars[sub] << endl;
43          sub += 1;
44     }    //end while
45  }  //end of displayArray function
```

your C++ development tool may not require this statement

Figure 11-63

17. See Figure 11-64. The changes are shaded in the figure.

```
 1  //TryThis17.cpp
 2  //Displays the total and average number of pounds
 3  //of coffee used during a 12-month period
 4  //Created/revised by <your name> on <current date>
 5
 6  #include <iostream>
 7  using namespace std;
 8
 9  //function prototype
10  double getTotal(double poundsUsed[], int numElements);
11  void getAverage(double totalLbs,
12                  int numElements,
13                  double &avg);
14
15  int main()
16  {
17     //declare array
18     double pounds[12] = {400.5, 450.0,
19          475.5, 336.5, 457.0, 325.0, 220.5,
20          276.0, 300.0, 320.5, 400.5, 415.0};
21     //declare variables
22     double total   = 0.0;
23     double average = 0.0;
24
25     //calculate the total and average pounds used
26     total = getTotal(pounds, 12);
27     getAverage(total, 12, average);
28
29     //display the total and average pounds used
30     cout << "Total pounds: " << total << endl;
31     cout << "Average pounds: " << average << endl;
32
33     system("pause");
34     return 0;
35  }  //end of main function
36
37  //*****function definitions*****
38  double getTotal(double poundsUsed[], int numElements)
39  {
40     double totalUsed = 0.0;     //accumulator
41
42     //accumulate the pounds used
43     for (int sub = 0; sub < numElements; sub += 1)
44          totalUsed += poundsUsed[sub];
45     //end for
46
47     return totalUsed;
48  }  //end of getTotal function
49
50  void getAverage(double totalLbs,
51                  int numElements,
52                  double &avg)
53  {
54     avg = totalLbs / numElements;
55  }  //end of getAverage function
```

your C++ development tool may not require this statement

Figure 11-64

CHAPTER 12

Two-Dimensional Arrays

After studying Chapter 12, you should be able to:

- Declare and initialize a two-dimensional array
- Enter data into a two-dimensional array
- Display the contents of a two-dimensional array
- Sum the values in a two-dimensional array
- Search a two-dimensional array
- Pass a two-dimensional array to a function

Using Two-Dimensional Arrays

As discussed in Chapter 11, an array is a group of related variables that have the same data type. Recall that the most commonly used arrays in business applications are one-dimensional and two-dimensional. You can visualize a one-dimensional array as a column of variables in memory, similar to the column of storage bins shown in Figure 11-1 in Chapter 11. A **two-dimensional array**, on the other hand, resembles a table in that the variables are in rows and columns. You can determine the number of variables in a two-dimensional array by multiplying the number of its rows by the number of its columns. An array that has four rows and three columns, for example, contains 12 variables. Each variable in a two-dimensional array is identified by a unique combination of two subscripts that the computer assigns to the variable when the array is created. The subscripts specify the variable's row and column positions in the array. Variables located in the first row in a two-dimensional array are assigned a row subscript of 0. Variables in the second row are assigned a row subscript of 1, and so on. Similarly, variables located in the first column in a two-dimensional array are assigned a column subscript of 0. Variables in the second column are assigned a column subscript of 1, and so on. You refer to each variable in a two-dimensional array by the array's name and the variable's row and column subscripts, with the row subscript listed first and the column subscript listed second. The row subscript is enclosed in a set of square brackets (`[]`) and so is the column subscript. For example, to refer to the variable located in the first row, first column in a two-dimensional array named `orders`, you use `orders[0][0]`—read "`orders` sub zero zero." Similarly, to refer to the variable located in the second row, third column, you use `orders[1][2]`. Notice that the subscripts are one number less than the row and column in which the variable is located; this is because the row and column subscripts start at 0 rather than at 1. You will find that the last row subscript in a two-dimensional array is always one number less than the number of rows in the array. Similarly, the last column subscript is always one number less than the number of columns in the array. Figure 12-1 illustrates the variables contained in the two-dimensional `orders` array using the storage bin analogy. The `rating` and `numSold` variables included in the figure are scalar variables.



Recall that a subscript also is called an index.

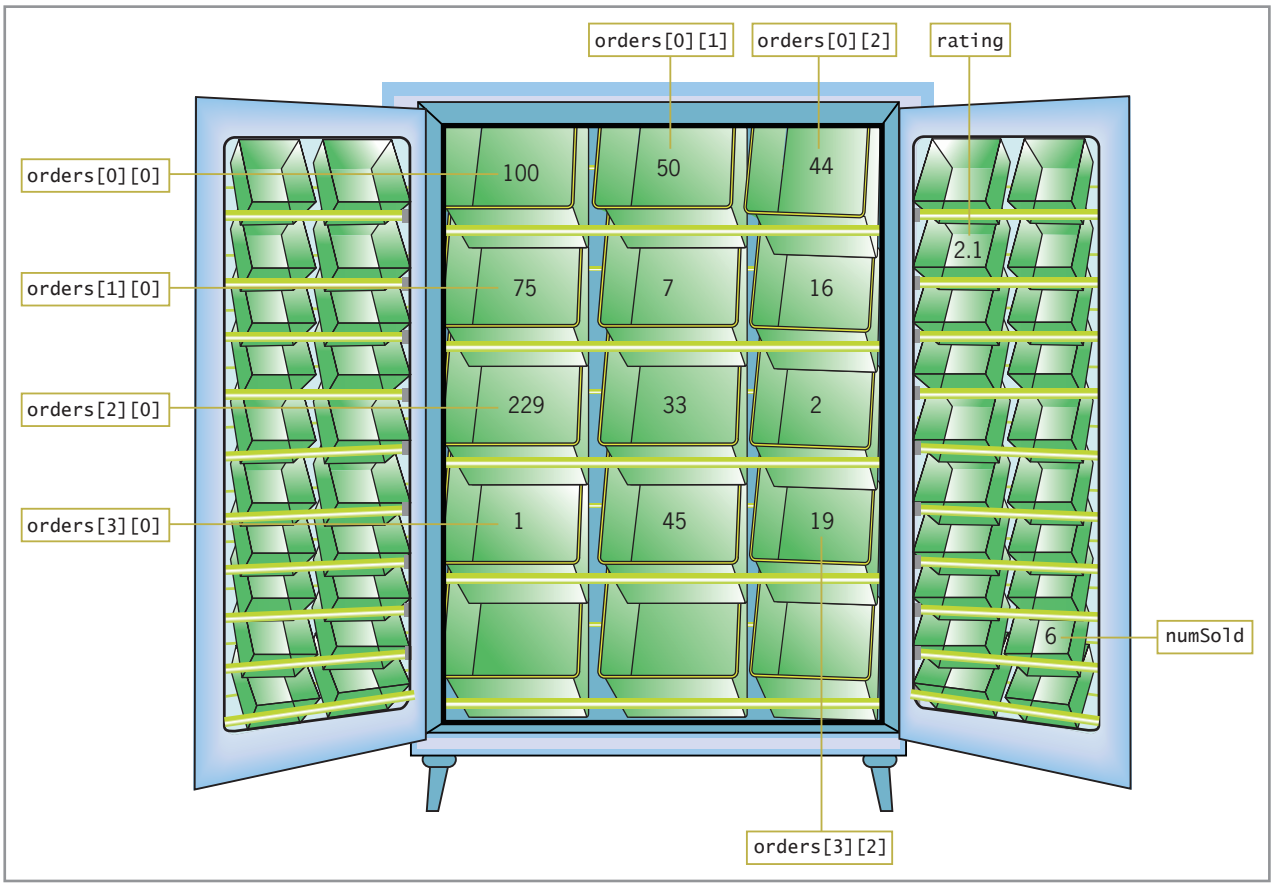


Figure 12-1 Names of some of the variables contained in the two-dimensional `orders` array

Figure 12-2 shows the problem specification and IPO chart information for the Caldwell Company's orders program, which uses a two-dimensional array to store the number of orders received from each of the company's four sales regions during the first three months of the year. Each row in the array represents a region, and each column represents a month. The program allows the user to enter the order amounts and then displays the amounts on the computer screen. Before you can code the orders program, you need to learn how to declare and initialize a two-dimensional array. You also need to learn how to store data in a two-dimensional array and display data from the array.

Problem specification
The Caldwell Company wants a program that allows the user to enter the number of orders received from each of its four sales regions during the first three months of the year. The order amounts should be stored in a two-dimensional `int` array that contains four rows and three columns. Each row represents a region, and each column represents a month. The program then should display the order amounts on the computer screen. The order amounts for Region 1 should be displayed first, followed by Region 2's order amounts, and so on.

Figure 12-2 Problem specification and IPO chart for the Caldwell Company's orders program *(continues)*

(continued)

Input	Processing	Output
number of orders (from each of 4 regions for 3 months)	Processing items: array (4 region rows, 3 month columns) region subscript counter (0 to 3) month subscript counter (0 to 2)	number of orders (from each of 4 regions for 3 months)

```
Algorithm:
1. repeat for (each of the 4 region rows)
        repeat for (each of the 3 month columns)
              enter the number of orders into
              the current array element, using
              the region subscript counter and
              the month subscript counter
        end repeat
   end repeat
2. repeat for (each of the 4 region rows)
        display the region number, use the region
        subscript counter + 1

        repeat for (each of the 3 month columns)
              display the month number, use the
              month subscript counter + 1

              display the number of orders
              stored in the current array element,
              using the region subscript counter and
               the month subscript counter
        end repeat
   end repeat
```

Figure 12-2 Problem specification and IPO chart for the Caldwell Company's orders program

Declaring and Initializing a Two-Dimensional Array

Before you can use a two-dimensional array in a program, you first must declare (create) it. It also is a good programming practice to initialize the array variables to ensure they will not contain garbage when the program is run. Recall that assigning initial values to an array is often referred to as populating the array. You should populate an array using values that have the same data type as the array. Figure 12-3 shows the syntax for declaring and initializing a two-dimensional array in C++. In the syntax, *arrayName* is the name of the array and *dataType* is the type of data the array variables (elements) will store. Recall that each of the elements in an array has the same data type. The *numberOfRows* and *numberOfColumns* items, each of which is enclosed in its own set of square brackets, are integers that specify the number of rows and columns, respectively, in the array. You can initialize the elements in a two-dimensional array by entering a separate *initialValues* section, enclosed in braces, for each row in the array. If the array has two rows, then the statement that declares and initializes the array can have a maximum of two *initialValues* sections. If the array has five rows, then the declaration statement can have a maximum of five *initialValues* sections.

Within the individual *initialValues* sections, you enter one or more values separated by commas. The maximum number of values you enter corresponds to the maximum number of columns in the array. If the array contains 10 columns, then you can include up to 10 values in each *initialValues* section. In addition to the set of braces that surrounds each individual *initialValues* section, notice in the syntax that a set of braces also surrounds all of the *initialValues* sections. Also shown in Figure 12-3 are examples of declaring and initializing two-dimensional arrays.

HOW TO Declare and Initialize a Two-Dimensional Array

Syntax

dataType arrayName[*numberOfRows*][*numberOfColumns*] =
{{*initialValues*}, {*initialValues*}, ...{*initialValues*}};

Example 1

```
char grades[3][2] = {{'C', 'A'}, {'B', 'C'}, {'D', 'B'}};
```

declares and initializes a three-row, two-column `char` array named `grades`

Example 2

```
int orders[4][3] = {0};
```

or

```
int orders[4][3] = {{0}, {0}, {0}, {0}};
```

or

```
int orders[4][3] = {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}};
```

declares and initializes a four-row, three-column `int` array named `orders`; each element is initialized to 0

Example 3

```
double prices[6][5] = {2.0};
```

declares and initializes a six-row, five-column `double` array named `prices`; the `prices[0][0]` element is initialized to 2.0; the other elements are initialized to 0.0

Figure 12-3 How to declare and initialize a two-dimensional array

The declaration statement in Example 1 in Figure 12-3 creates a two-dimensional `char` array named `grades`. The `grades` array contains three rows and two columns. The statement initializes the first row in the array to the grades C and A, the second row to the grades B and C, and the third row to the grades D and B, as illustrated in Figure 12-4.

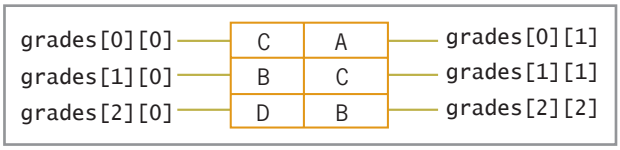


Figure 12-4 Illustration of the two-dimensional `grades` array

You can use any of the three statements shown in Example 2 in Figure 12-3 to declare the two-dimensional `orders` array and initialize the variables

(elements) in its four rows and three columns to the number 0. When you don't provide an initial value for each of the elements in an `int` array, most C+ compilers initialize the uninitialized elements to the integer 0. The statement shown in Example 3 in Figure 12-3 declares a two-dimensional `double` array named `prices`; the array contains six rows and five columns. The statement initializes the element located in the first row, first column of the array to 2.0. The remaining elements will be initialized to the `double` number 0.0 by most C++ compilers. Keep in mind that if you inadvertently provide more *initialValues* sections than the number of rows in the array, or if you provide more values in an *initialValues* section than the number of columns in the array, most C++ compilers will display the error message "too many initializers" when you attempt to compile the program. However, not all C++ compilers display a message when this error occurs. Rather, some compilers store the extra values in memory locations adjacent to, but not reserved for, the array. Therefore, you always should be careful to provide the appropriate number of *initialValues* sections and the appropriate number of values in each section.

Entering Data into a Two-Dimensional Array

As you can with one-dimensional arrays, you can use either an assignment statement or the extraction operator to enter data into the elements of a two-dimensional array. Figure 12-5 shows the syntax of an assignment statement that accomplishes the entry task. The *arrayName*[*rowSubscript*][*columnSubscript*] section of the syntax represents the name and subscripts of the element to which you want the *expression* (data) assigned. The *expression* can include any combination of constants, variables, and operators. The data type of the *expression* must match the data type of the array element to which the *expression* is assigned. If the data types do not match, an implicit type conversion will occur and may result in incorrect output. Also included in Figure 12-5 are examples of assignment statements that assign data to the elements in various arrays. The arrays were declared earlier in Figure 12-3. The assignment statement in Example 1 assigns the letter F to the element located in the second row, first column in the `grades` array, replacing the letter B that was stored in the element when the array was declared. The code in Example 2 assigns the integer 0 to each of the 12 elements in the `orders` array and provides another means of initializing the array. Notice that the code uses two loops to access each element in the array. One of the loops keeps track of the row subscript, while the other loop keeps track of the column subscript. The code assigns the integer 0 to the array, row by row. In other words, it assigns 0 to each element in the first row before assigning 0 to each element in the second row and so on. The code in Example 3 assigns a new price to each of the elements in the `prices` array. The new price is calculated by the assignment statement within the nested loop. The statement multiplies the old price by the contents of the `INCREASE` named constant and then assigns the result to the current element in the array. Like the code in Example 2, the code in Example 3 needs to use two loops to access each element in the array. However, unlike the code in Example 2, the code in Example 3 assigns values to the array, column by column, rather than row by row. This means that the code will assign values to each element in the first column before assigning values to each element in the second column and so on.



Recall that you can determine the number of elements in a two-dimensional array by multiplying the number of its rows by the number of its columns.

HOW TO Use an Assignment Statement to Assign Data to a Two-Dimensional Array

Syntax

arrayName[*rowSubscript*][*columnSubscript*] = *expression*;

Example 1

```
grades[1][0] = 'F';
```

assigns the letter F to the element located in the second row, first column in the `grades` array

Example 2

```
for (int row = 0; row < 4; row += 1)
      for (int column = 0; column < 3; column += 1)
            orders[row][column] = 0;
      //end for
//end for
```

assigns the integer 0 to each element in the four-row, three-column `orders` array, row by row; provides another means of initializing the array

Example 3

```
int row = 0;
int column = 0;
double oldPrice = 0.0;
const double INCREASE = 1.15;
while (column < 5)
{
      while (row < 6)
      {
            cout << "Price: ";
            cin >> oldPrice;
            prices[row][column] = oldPrice * INCREASE;
            row += 1;
      }     //end while
      column += 1;
      row = 0;
}     //end while
```

assigns the new price to each element in the six-row, five-column `prices` array, column by column; the new price is calculated by multiplying the old price by the value stored in the `INCREASE` named constant



You also can use the C++ increment operator (`++`) to add 1 to a variable. For instance, you can use `row++` and `column++` in Example 2, and `row++;` and `column++;` in Example 3.



Be sure to always use valid row and column subscripts when referring to an element in a two-dimensional array. If the compiler encounters an invalid subscript, it will display an error message and the program will end abruptly.

Figure 12-5 How to use an assignment statement to assign data to a two-dimensional array

As already mentioned, you also can use the extraction operator to store data in an element in a two-dimensional array. Figure 12-6 shows the syntax and examples of doing this. (The arrays in Figure 12-6 were declared earlier in Figure 12-3.) The `cin` statement in Example 1 stores the user's entry in the element located in the third row, second column in the `grades` array, replacing the element's existing data. The code in Example 2 contains two `for` loops. The instructions in the outer loop will be repeated once for each region, while the instructions in the nested loop will be repeated once for

each month within each region. The `cout` statement in the nested loop prompts the user to enter the number of orders for the current region and month. The `cin` statement stores the user's response in the current element in the `orders` array. The responses will be stored, row by row, in the array. In other words, the monthly sales for Region 1 will be stored before the monthly sales for Region 2 and so on. Example 3 contains an outer `while` loop and a nested `for` loop. The `while` loop repeats its instructions for each column in the array, and the `for` loop repeats its instructions for each row in the array. The `cout` statement in the `for` loop prompts the user to enter a price, and the `cin` statement stores the user's response in the current element in the `prices` array. The responses will be stored, column by column, in the array.

HOW TO Use the Extraction Operator to Store Data in a Two-Dimensional Array

Syntax

```
cin >> arrayName[subscript][subscript];
```

Example 1

```
cin >> grades[2][1];
```

stores the user's entry in the element located in the third row, second column in the `grades` array

Example 2

```
for (int region = 0; region < 4; region += 1)
      for (int month = 0; month < 3; month += 1)
      {
            cout << "Number of orders for Region "
                 << region + 1 << ", Month "
                 << month + 1 << ": ";
            cin >> orders[region][month];
      }     //end for
//end for
```

you also can use `region++`

you also can use `month++`

stores the user's entries in the four-row, three-column `orders` array, region (row) by region (row)

Example 3

```
int column = 0;
while (column < 5)
{
      for (int row = 0; row < 6; row += 1)
      {
            cout << "Price: ";
            cin >> prices[row][column];
      }     //end for
      column += 1;
}     //end while
```

you also can use `row++`

you also can use `column++;`

stores the user's entries in the six-row, five-column `prices` array, column by column

Figure 12-6 How to use the extraction operator to store data in a two-dimensional array

Displaying the Contents of a Two-Dimensional Array

To display the contents of a two-dimensional array, you need to access each of its elements. You do this using two counter-controlled loops: one to keep track of the row subscript and the other to keep track of the column subscript. Figure 12-7 shows examples of loops you can use to display the contents of the arrays declared earlier in Figure 12-3. Example 1 uses two `while` loops to display the contents of the `grades` array, column by column. Recall that the `grades` array contains three rows and two columns. Example 2 uses two `for` loops to display the contents of the four-row, three-column `orders` array, region (row) by region (row). Example 3 uses both a `do while` loop and a `for` loop to display the contents of the six-row, five-column `prices` array, row by row.



You also can use `row++;` and `column++;` in Example 1, `region++` and `month++` in Example 2, and `column++` and `row++;` in Example 3.

HOW TO Display the Contents of a Two-Dimensional Array

Example 1

```
int row = 0;
int column = 0;
while (column < 2)
{
      while (row < 3)
      {
            cout << grades[row][column] << endl;
            row += 1;
      }     //end while
      column += 1;
      row = 0;
}     //end while
```

displays the contents of the three-row, two column `grades` array, column by column

Example 2

```
for (int region = 0; region < 4; region += 1)
      for (int month = 0; month < 3; month += 1)
            cout << orders[region][month] << endl;
      //end for
//end for
```

displays the contents of the four-row, three column `orders` array, region (row) by region (row)

Example 3

```
int row = 0;
do    //begin loop
{
      for (int column = 0; column < 5; column += 1)
            cout << prices[row][column] << endl;
      //end for
      row += 1;
} while (row < 6);
```

displays the contents of the six-row, five-column `prices` array, row by row

Figure 12-7 How to display the contents of a two-dimensional array

Now that you know how to declare and initialize a two-dimensional array, as well as how to store data in the array and display data from the array, you can code the Caldwell Company's orders program.

Coding the Caldwell Company's Orders Program

Earlier, in Figure 12-2, you viewed the problem specification and IPO chart for the Caldwell Company's orders program. Figure 12-8 shows the IPO chart information along with the corresponding C++ instructions. Figure 12-9 shows the code for the entire program, and Figure 12-10 shows a sample run of the program.

IPO chart information	C++ instructions
Input	
number of orders (from each of 4 regions for 3 months)	the orders will be entered into the array
Processing	
array (4 region rows, 3 month columns)	`int orders[4][3] = {0};`
region subscript counter (0 to 3)	declared and initialized in the for clause
month subscript counter (0 to 2)	declared and initialized in the for clause
Output	
number of orders (from each of 4 regions for 3 months)	displayed from the array by the for loops
Algorithm	

1. repeat for (each of the 4 region rows)
repeat for (each of the 3 month columns)
enter the number of orders into the current array element, using the region subscript counter and the month subscript counter
end repeat
end repeat
	`for (int region = 0; region < 4; region += 1)`
	`for (int month = 0; month < 3; month += 1)`
	`{`
	`cout << "Number of orders for Region " << region + 1 << ", Month " << month + 1 << ": ";`
	`cin >> orders[region][month];`
	`} //end for`
	`//end for`

2. repeat for (each of the 4 region rows)
display the region number, use the region subscript counter + 1
repeat for (each of the 3 month columns)
display the month number, use the month subscript counter + 1
	`for (int region = 0; region < 4; region += 1)`
	`{`
	`cout << "Region " << region + 1 << ": " << endl;`
	`for (int month = 0; month < 3; month += 1)`
	`{`
	`cout << " Month " << month + 1 << ": ";`

Figure 12-8 IPO chart information and C++ instructions for the Caldwell Company's orders program *(continues)*

(continued)

```
      display the number of orders                cout << orders[region][month]
      stored in the current array                 << endl;
      element, using the region
      subscript counter and the
      month subscript counter
   end repeat                                   } //end for
end repeat                                    } //end for
```

Figure 12-8 IPO chart information and C++ instructions for the Caldwell Company's orders program



You also can use `region++` in Lines 14 and 25, and `month++` in Lines 15 and 29.

```
 1  //Caldwell Company.cpp
 2  //Displays the contents of a two-dimensional array
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  using namespace std;
 7
 8  int main()
 9  {
10     //declare and initialize array
11     int orders[4][3] = {0};                    array declaration
12
13     //enter data into the array
14     for (int region = 0; region < 4; region += 1)
15          for (int month = 0; month < 3; month += 1)
16          {
17               cout << "Number of orders for Region "
18                    << region + 1 << ", Month "
19                    << month + 1 << ": ";
20               cin >> orders[region][month];
21          }   //end for
22     //end for                                 stores data in the array
23
24     //display the contents of the array
25     for (int region = 0; region < 4; region += 1)
26     {
27          cout << "Region " << region + 1
28               << ": " << endl;
29          for (int month = 0; month < 3; month += 1)
30          {
31               cout << " Month " << month + 1
32                    << ": ";
33               cout << orders[region][month] << endl;
34          }   //end for
35     }    //end for                             displays the contents of the array
36
37     system("pause");                           your C++ development tool may not require this statement
38     return 0;
39  }  //end of main function
```

Figure 12-9 Caldwell Company's orders program

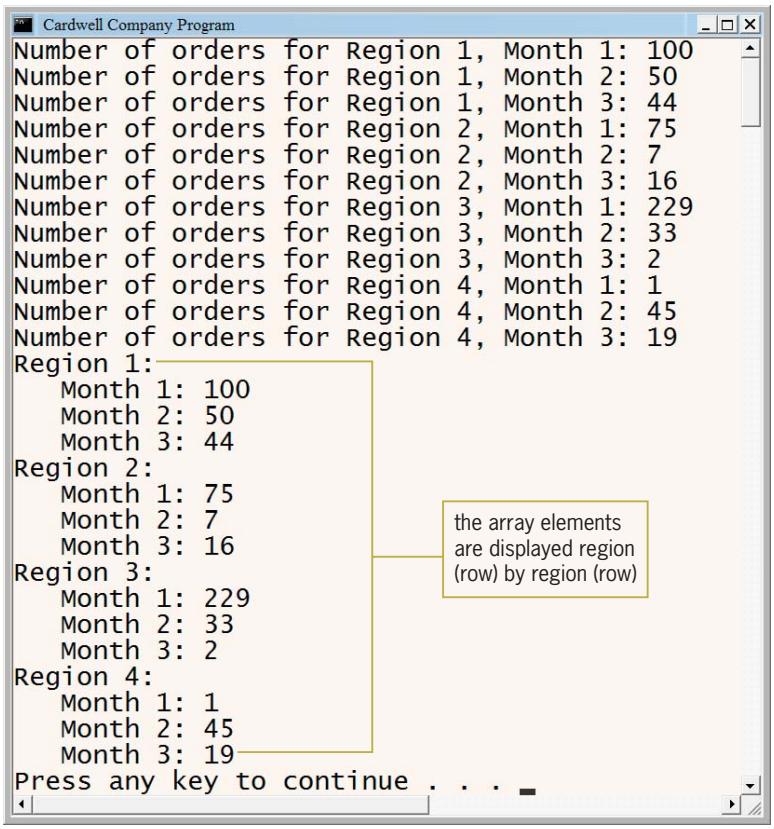

Cardwell Company Program
Number of orders for Region 1, Month 1: 100
Number of orders for Region 1, Month 2: 50
Number of orders for Region 1, Month 3: 44
Number of orders for Region 2, Month 1: 75
Number of orders for Region 2, Month 2: 7
Number of orders for Region 2, Month 3: 16
Number of orders for Region 3, Month 1: 229
Number of orders for Region 3, Month 2: 33
Number of orders for Region 3, Month 3: 2
Number of orders for Region 4, Month 1: 1
Number of orders for Region 4, Month 2: 45
Number of orders for Region 4, Month 3: 19
Region 1:
Month 1: 100
Month 2: 50
Month 3: 44
Region 2:
Month 1: 75
Month 2: 7
Month 3: 16
Region 3:
Month 1: 229
Month 2: 33
Month 3: 2
Region 4:
Month 1: 1
Month 2: 45
Month 3: 19
Press any key to continue . . .
the array elements are displayed region (row) by region (row)

Figure 12-10 Sample run of the Caldwell Company's orders program

Mini-Quiz 12-1



The answers to Mini-Quiz questions are located in Appendix A.

1. Which of the following declares a four-row, two column `int` array named `quantities` and initializes each of its 8 elements to the number 0?

   a. `int quantities[2][4] = {0};`

   b. `int quantities[4][2] = {0};`

   c. `int quantities{2}{4} = [0];`

   c. `int quantities{4}{2} = [0];`

2. How many elements are contained in a five-row, four-column array?

3. What is the name of the first element in the `quantities` array from Question 1?

3. What is the name of the last element in the `quantities` array from Question 1?

4. Write a C++ statement that assigns the number 5 to the element located in the second column, first row in the `quantities` array from Question 1.

Accumulating the Values Stored in a Two-Dimensional Array

Figure 12-11 shows the problem specification, IPO chart information, and C++ instructions for the Jenko Booksellers program. The program uses a two-dimensional array to store the sales made in each of the company's three bookstores. The array contains three rows and two columns. The first column in the array contains the sales amounts for paperback books sold in each of the three stores. The second column contains the sales amounts for hardcover books. The program calculates the total sales by accumulating the amounts stored in the array. It then displays the total sales on the computer screen. Figure 12-12 shows the code for the entire program, and Figure 12-13 shows the result of running the program.

Problem specification

Jenko Booksellers wants a program that calculates and displays the total of its previous month's sales. The sales amounts, which are shown here, are stored in a two-dimensional `double` array named `sales`. The `sales` array contains three rows and two columns. The first column contains the sales amounts for paperback books sold in each of the three stores. The second column contains the sales amounts for hardcover books sold in each of the three stores.

	Paperback sales ($)	Hardcover sales ($)
Store 1	1200.33	2350.75
Store 2	3677.80	2456.05
Store 3	750.67	1345.99

IPO chart information	C++ instructions
Input	
store sales (made in each of 3 stores for 2 types of books)	the sales are stored in the array
Processing	
array (3 store rows, 2 book columns)	`double sales[3][2] = {{1200.33, 2350.75}, {3677.80, 2456.05}, {750.67, 1345.99}};`
store subscript counter (0 to 2)	declared and initialized in the for clause
book subscript counter (0 to 1)	declared and initialized in the for clause
Output	
total sales (accumulator)	`double total = 0.0;`

Figure 12-11 Problem specification, IPO chart information, and C++ instructions for the Jenko Booksellers program *(continues)*

(continued)

Algorithm

```
1. repeat for (each of the                 for (int store = 0; store < 3;
   3 store rows)                           store += 1)
      repeat for (each of the                 for (int book = 0; book < 2;
      2 book columns)                         book += 1)
         add the store sales from                total += sales[store][book];
         the current array element,
         using the store subscript
         counter and the book
         subscript counter, to the
         total sales
      end repeat                              //end for
   end repeat                             //end for

2. display the total sales                cout << "Total sales: $" << total
                                          << endl;
```

Figure 12-11 Problem specification, IPO chart information, and C++ instructions for the Jenko Booksellers program

```
 1  //Jenko Booksellers.cpp - displays the total sales
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <iomanip>
 6  using namespace std;
 7
 8  int main()
 9  {
10     //declare array and variable
11     double sales[3][2] = {{1200.33, 2350.75},
12                           {3677.80, 2456.05},
13                           {750.67, 1345.99}};
14     double total = 0.0;     //accumulator
15
16     //accumulate sales
17     for (int store = 0; store < 3; store += 1)
18          for (int book = 0; book < 2; book += 1)
19               total += sales[store][book];
20          //end for
21     //end for
22
23     //display total sales
24     cout << fixed << setprecision(2);
25     cout << "Total sales: $" << total << endl;
26
27     system("pause");
28     return 0;
29  }  //end of main function
```

array declaration

accumulates the sales stored in the array

your C++ development tool may not require this statement



You also can use `store++` in Line 17 and `book++` in Line 18.

Figure 12-12 Jenko Booksellers program

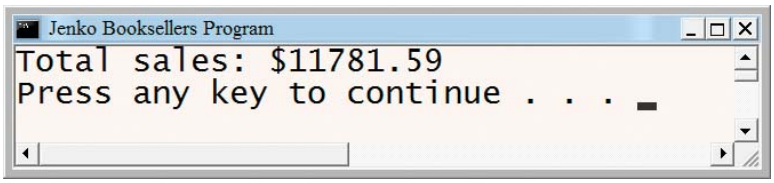


Figure 12-13 Result of running the Jenko Booksellers program

Searching a Two-Dimensional Array

Figure 12-14 shows the problem specification, IPO chart information, and C++ instructions for the Wilson Company's pay rate program. The program uses a four-row, two-column array to store the company's four pay codes and their corresponding pay rates. The pay codes are stored in the first column of each row in the array. The pay rate associated with each code is stored in the same row as its pay code, but in the second column. The program gets a pay code from the user and then searches for the pay code in the array's first column. If it finds the pay code, the program displays the corresponding pay rate from the second column; otherwise, it displays the "Invalid pay code" message. For example, when the user enters a pay code of 6, the program displays a pay rate of 14, as shown in Figure 12-15. This is because the number 6 is contained in the `codesAndRates[2][0]` element, and its corresponding pay rate is contained in the `codesAndRates[2][1]` element . Notice that the pay code and its associated pay rate are contained in the same row but in different columns. However, as Figure 12-16 shows, the program displays the "Invalid pay code" message when the user enters the number 5 as the pay code. This is because the number 5 does not appear in the first column in the array. Figure 12-17 shows the code for the entire program

Problem specification

Wilson Company wants a program that displays the pay rate corresponding to the pay code entered by the user. The pay codes and rates, which are listed here, are stored in a two-dimensional `int` array named `codesAndRates`. The `codesAndRates` array contains four rows and two columns. The first column contains the company's four different pay codes, and the second column contains each code's corresponding rate.

Pay code	Pay rate
3	8
7	10
6	14
9	20

IPO chart information	**C++ instructions**
Input	
4 pay codes and their corresponding pay rates	the pay codes and pay rates are stored in the array
pay code to search for	`int payCode = 0;`

Figure 12-14 Problem specification, IPO chart information, and C++ instructions for the Wilson Company program *(continues)*

(continued)

Processing

array (4 rows, 2 columns)	`int codesAndRates[4][2] =`
	`{{3, 8},`
	`{7, 10},`
	`{6, 14},`
	`{9, 20}};`

row subscript counter (0 to 3)	`int rowSubscript = 0;`

Output

pay rate	displayed from the array

Algorithm

1. enter the pay code to search for	`cout << "Pay code (3, 7, 6,`
	`or 9). " << endl;`
	`cout << "Enter a negative`
	`number to end: ";`
	`cin >> payCode;`

2. repeat while (the pay code to search for is greater than or equal to 0)	`while (payCode >= 0)` `{`
assign 0 to the row subscript counter to begin searching the array with the first row	`rowSubscript = 0;`

repeat while (the row subscript counter is less than or equal to 3 and the pay code stored in the first column of the current row is not the pay code to search for)	`while (rowSubscript <= 3 &&`
	`codesAndRates[rowSubscript][0]`
	`!= payCode)`

add 1 to the row subscript counter to continue the search with the next row	`rowSubscript += 1;`
end repeat	`//end while`
if (the row subscript counter is less than or equal to 3)	`if (rowSubscript <= 3)`

display the pay code and pay rate located in the same row as the pay code, but in the second column	`cout << "Pay rate`
	`for pay code " <<`
	`codesAndRates[rowSubscript][0]`
	`<< ": $" <<`
	`codesAndRates[rowSubscript][1]`
	`<< endl << endl;`

else	`else`
display "Invalid pay code"	`cout << "Invalid pay code"` `<< endl << endl;`
end if	`//end if`

enter the pay code to search for	`cout << "Pay code (3, 7, 6,`
	`or 9). " << endl;`
	`cout << "Enter a negative`
	`number to end: ";`
	`cin >> payCode;`

end while	`}  //end while`

Figure 12-14 Problem specification, IPO chart information, and C++ instructions for the Wilson Company program

pay code and pay rate from `codesAndRates[2][0]` and `codesAndRates[2][1]`, respectively

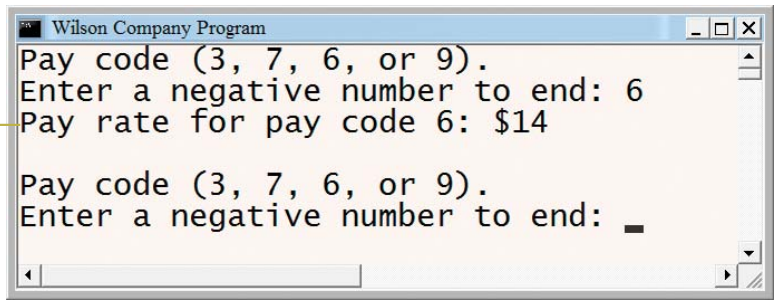

Wilson Company Program
Pay code (3, 7, 6, or 9).
Enter a negative number to end: 6
Pay rate for pay code 6: $14
Pay code (3, 7, 6, or 9).
Enter a negative number to end:

Figure 12-15 Sample run of the Wilson Company program

displayed when the pay code is not valid

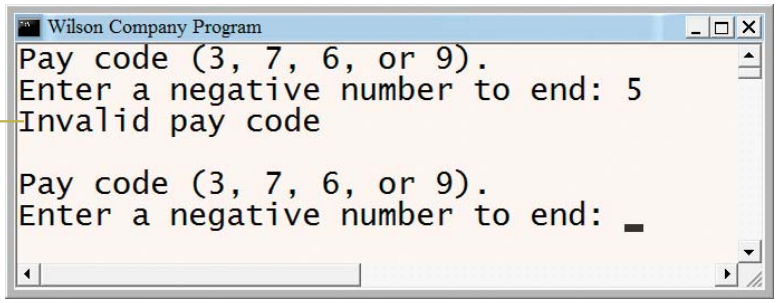

Wilson Company Program
Pay code (3, 7, 6, or 9).
Enter a negative number to end: 5
Invalid pay code
Pay code (3, 7, 6, or 9).
Enter a negative number to end:

Figure 12-16 Another sample run of the Wilson Company program

```
 1  //Wilson Company.cpp - displays the pay rate
 2  //corresponding to the pay code entered by the user
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  using namespace std;
 7
 8  int main()
 9  {
10     //declare array and variables
11     int codesAndRates[4][2] = {{3, 8},
12                                {7, 10},
13                                {6, 14},
14                                {9, 20}};
15     int payCode      = 0;
16     int rowSubscript = 0;
17
18     //get pay code
19     cout << "Pay code (3, 7, 6, or 9). " << endl;
20     cout << "Enter a negative number to end: ";
21     cin >> payCode;
22
23     while (payCode >= 0)
24     {
25          //search each row in the array, looking
26          //for the pay code in the first column
27          //continue the search while there are
28          //array elements to search and the pay
29          //code has not been found
```

Figure 12-17 Wilson Company program *(continues)*

(continued)

```
30          rowSubscript = 0;
31          while (rowSubscript <= 3
32               && codesAndRates[rowSubscript][0]
33               != payCode)
34                    rowSubscript += 1;
35          //end while
36
37          //if the pay code was found, display the
38          //pay rate located in the same row but in
39          //the second column in the array
40          if (rowSubscript <= 3)
41               cout << "Pay rate for pay code "
42                    << codesAndRates[rowSubscript][0]
43                    << ": $"
44                    << codesAndRates[rowSubscript][1]
45                    << endl << endl;
46          else
47               cout << "Invalid pay code" << endl << endl;
48          //end if
49
50          //get pay code
51          cout << "Pay code (3, 7, 6, or 9). " << endl;
52          cout << "Enter a negative number to end: ";
53          cin >> payCode;
54     }    //end while
55
56     system("pause");
57     return 0;
58 }   //end of main function
```

you also can use `rowSubscript++;`

your C++ development tool may not require this statement

Figure 12-17 Wilson Company program

You will desk-check the program in Figure 12-17 to observe how it searches the array. The declaration statements on Lines 11 through 16 create and initialize the two-dimensional `codesAndRates` array and two variables named `payCode` and `rowSubscript`. The `cout` statements on Lines 19 and 20 prompt the user to enter a pay code, and the `cin` statement on Line 21 stores the user's response in the `payCode` variable. Assume that the user enters the number 6 as the pay code. Figure 12-18 shows the desk-check table after the statements on Lines 11 through 21 are processed.

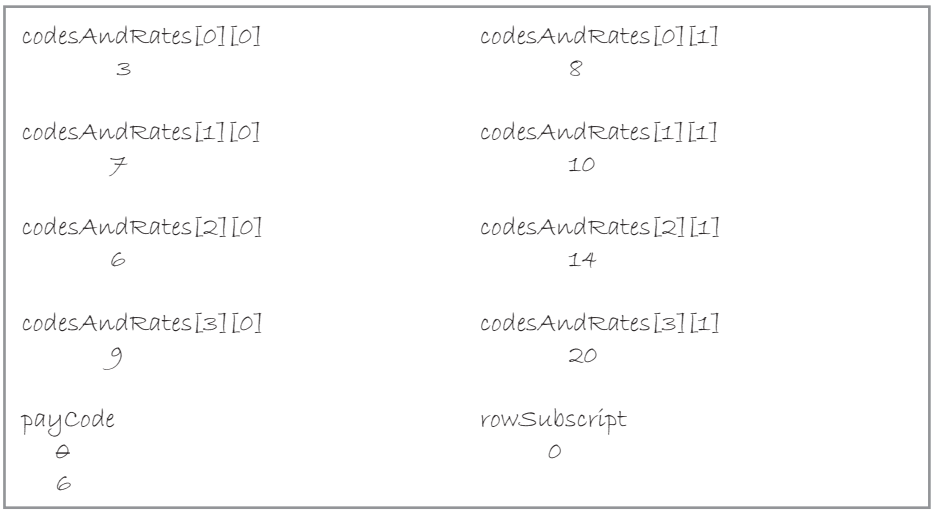


Figure 12-18 Desk-check table after the statements on Lines 11 through 21 are processed

The `while` clause in the program's outer loop is processed next. The clause's condition evaluates to true, because the value in the `payCode` variable is greater than or equal to the number 0. Therefore, the computer processes the outer loop's instructions. The instructions on Lines 30 through 35 in the outer loop search for the pay code in the first column of the array; study these instructions closely. The instruction on Line 30 assigns the number 0 to the `rowSubscript` variable to ensure that the search will begin in the first row. The `while` clause on Lines 31 through 33 marks the beginning of a nested loop. The clause's compound condition evaluates to true, because both of its sub-conditions evaluate to true. The first sub-condition determines whether the value in the `rowSubscript` variable is less than or equal to 3 (the highest row subscript in the array). The sub-condition evaluates to true, because the `rowSubscript` variable contains the number 0. The second sub-condition compares the value in the `codesAndRates[rowSubscript][0]` element with the value in the `payCode` variable. This sub-condition also evaluates to true, because the value in the `codesAndRates[0][0]` element (3) is not equal to the value in the `payCode` variable (6). As a result, the `rowSubscript += 1;` statement on Line 34 in the nested loop is processed. The statement adds the number 1 to the contents of the `rowSubscript` variable, giving 1. Incrementing the `rowSubscript` variable by 1 allows the computer to search the next row in the array. Figure 12-19 shows the desk-check table after the nested loop is processed the first time.

codesAndRates[0][0]	codesAndRates[0][1]
3	8
codesAndRates[1][0]	codesAndRates[1][1]
7	10
codesAndRates[2][0]	codesAndRates[2][1]
6	14
codesAndRates[3][0]	codesAndRates[3][1]
9	20
payCode	rowSubscript
0	0
6	0
	1

Figure 12-19 Desk-check table after the nested loop is processed the first time

The compound condition in the nested loop's `while` clause is evaluated again. The compound condition evaluates to true, because the value in the `rowSubscript` variable (1) is less than or equal to 3 and (at the same time) the value in the `codesAndRates[1][0]` element (7) is not equal to the value in the `payCode` variable (6). Therefore, the `rowSubscript += 1;` statement on Line 34 increments the `rowSubscript` variable by 1; the result is 2. Figure 12-20 shows the desk-check table after the nested loop is processed the second time.

codesAndRates[0][0]	codesAndRates[0][1]
3	8
codesAndRates[1][0]	codesAndRates[1][1]
7	10
codesAndRates[2][0]	codesAndRates[2][1]
6	14
codesAndRates[3][0]	codesAndRates[3][1]
9	20
payCode	rowSubscript
0	0
6	0
	1
	2

Figure 12-20 Desk-check table after the nested loop is processed the second time

The compound condition in the nested loop's `while` clause is evaluated once again. This time, the compound condition evaluates to false, because the value in the `codesAndRates[2][0]` element (6) *is* equal to the value in the `payCode` variable (6). At this point, the nested loop ends and processing continues with the `if` clause on Line 40. The `if` clause's condition

determines whether the value in the `rowSubscript` variable is less than or equal to 3 (the highest row subscript in the array). The condition evaluates to true, because the value in the `rowSubscript` variable is 2. Therefore, the computer processes the `cout` statement that appears on Lines 40 through 45. The statement displays a message containing the pay code stored in the `codesAndRates[2][0]` element and the pay rate stored in the `codesAndRates[2][1]` element, as shown earlier in Figure 12-15. After the message is displayed, the `if` statement ends. The `cout` statements on Lines 51 and 52 are processed next. Those statements prompt the user to enter another pay code. The `cin` statement on Line 53 then stores the user's response in the `payCode` variable. Assume that the user enters the number 5. The computer evaluates the condition in the outer loop's `while` clause next. The condition evaluates to true, because the value in the `payCode` variable (5) is greater than or equal to the number 0. Therefore, the computer processes the outer loop's instructions. The instruction on Line 30 in the outer loop assigns the number 0 to the `rowSubscript` variable to ensure that this new search will begin in the first row. The `while` clause on Lines 31 through 33 tells the computer to process the `rowSubscript += 1;` statement as long as the value in the `rowSubscript` variable is less than or equal to 3 (the highest row subscript in the array) and the current array element does not contain the pay code stored in the `payCode` variable. The nested loop will stop when either of the following occurs: the `rowSubscript` variable contains the number 4 (which indicates that the nested loop reached the end of the array without finding the pay code) or the pay code is located in the array's first column. Figure 12-21 shows the desk-check table after the nested loop ends. Unlike the nested loop in the previous search, which stopped when the pay code of 6 was located, the nested loop in this search stops when the `rowSubscript` variable contains the number 4.

```
codesAndRates[0][0]          codesAndRates[0][1]
        3                            8

codesAndRates[1][0]          codesAndRates[1][1]
        7                            10

codesAndRates[2][0]          codesAndRates[2][1]
        6                            14

codesAndRates[3][0]          codesAndRates[3][1]
        9                            20

payCode                      rowSubscript
  0                            0
  6                            0
  5                            1
                               2
                               0
                               1
                               2
                               3
                               4
```

Figure 12-21 Desk-check table after the nested loop ends during the second search

When the nested loop ends, processing continues with the `if` clause on Line 40. The `if` clause's condition determines whether the value in the `rowSubscript` variable is less than or equal to 3 (the highest row subscript in the array). The condition evaluates to false, because the value in the `rowSubscript` variable is 4. This indicates that the nested loop stopped processing because it reached the end of the array's first column without finding the pay code. Therefore, the computer processes the `cout` statement on Line 47. The statement displays the "Invalid pay code" message, as shown earlier in Figure 12-16. After the message is displayed, the `if` statement ends. Next, the `cout` statements on Lines 51 and 52 prompt the user to enter another pay code, and the `cin` statement on Line 53 stores the user's response in the `payCode` variable. Assume that the user enters the number -1. The computer evaluates the condition in the outer loop's `while` clause next. The condition evaluates to false, because the value in the `payCode` variable (-1) is not greater than or equal to the number 0. As a result, the outer loop ends and the computer processes the `system("pause");` and `return 0;` statements on Lines 56 and 57. After the `return` statement is processed, the program ends, and the computer removes the array and scalar variables from internal memory.

Passing a Two-Dimensional Array to a Function

Figure 12-22 shows a modified version of the Caldwell Company's orders program, which you viewed earlier in Figure 12-9. In the modified version, the `main` function passes the `orders` array to a program-defined void function named `displayArray`. Study closely the `displayArray` function prototype, function call, and function header; each is shaded in the figure. The function call appears on Line 28 and passes one actual argument to the `displayArray` function: the `orders` array. Like one-dimensional arrays, two-dimensional arrays are passed automatically *by reference*. The `displayArray` function prototype and function header appear on Lines 9 and 35, respectively; both contain one formal parameter: `nums[4][3]`. The first set of square brackets that follows the formal parameter's name contains the number of rows in the array; the second set contains the number of columns. Recall that the formal parameter's name is optional in the prototype. Therefore, you also could write the formal parameter in the function prototype as `int [4][3]`.



You learned about passing *by value* and *by reference* in Chapters 9 and 10.



When passing a two-dimensional array, the first set of square brackets in its corresponding formal parameter also can be empty, like this: `[]`. This concept is covered in Computer Exercise 16 at the end of the chapter.



You also can use region++ in Lines 17 and 37, and month++ in Lines 18 and 41.

```
1  //Modified Caldwell Company.cpp
2  //Displays the contents of a two-dimensional array
3  //Created/revised by <your name> on <current date>
4
5  #include <iostream>
6  using namespace std;
7
8  //function prototype
9  void displayArray(int nums[4][3]);
10
11 int main()
12 {
13    //declare and initialize array
14    int orders[4][3] = {0};
15
16    //enter data into the array
17    for (int region = 0; region < 4; region += 1)
18         for (int month = 0; month < 3; month += 1)
19         {
20              cout << "Number of orders for Region "
21                   << region + 1 << ", Month "
22                   << month + 1 << ": ";
23              cin >> orders[region][month];
24         }    //end for
25    //end for
26
27    //display the contents of the array
28    displayArray(orders);
29
30    system("pause");
31    return 0;
32 }  //end of main function
33
34 //*****function definitions*****
35 void displayArray(int nums[4][3])
36 {
37    for (int region = 0; region < 4; region += 1)
38    {
39         cout << "Region " << region + 1
40              << ": " << endl;
41         for (int month = 0; month < 3; month += 1)
42         {
43              cout << " Month " << month + 1
44                   << ": ";
45              cout << nums[region][month] << endl;
46         }  //end for
47    }    //end for
48 }  //end of displayArray function
```

your C++ development tool may not require this statement

Figure 12-22 Modified Caldwell Company orders program

Mini-Quiz 12-2



The answers to Mini-Quiz questions are located in Appendix A.

1. Which of the following increases the `total` variable by the contents of the element located in the second row, first column of the `purchases` array? The variable and array have the `int` data type.

   a. `purchases[2][1] += total;`

   b. `purchases[1][2] += total;`

   c. `total += purchases[2][1];`

   d. `total += purchases[1][0];`

2. Which of the following `if` clauses determines whether the value stored in the fourth column, third row in the `scores` array is greater than 25? The array has the `int` data type.

   a. `if (scores[3, 4] > 25)`

   b. `if (scores[4, 3] > 25)`

   c. `if (scores[2][3] > 25)`

   d. `if (scores[3][4] > 25)`

3. Write a C++ statement that multiplies the contents of the element located in the first row, second column in the `sales` array by .15 and then stores the result in the `bonus` variable. The `sales` array and `bonus` variable have the `double` data type.

4. Which of the following `if` clauses determines whether an `int` variable named `row` contains a valid subscript for the `scores` array? The array has 10 rows and 20 columns.

   a. `if (row >= 0 && row < 10)`

   b. `if (row >= 0 && row <= 10)`

   c. `if (row >= 0 && row < 9)`

   d. `if (row > 0 && row <= 10)`



LAB 12-1 Stop and Analyze



The answers to the labs are located in Appendix A.

Study the program shown in Figure 12-23, and then answer the questions. The `company` array contains the amounts the company sold both domestically and internationally during the months of January through June. The first row contains the domestic sales for the three months. The second row contains the international sales during the same period.

```
1  //Lab12-1.cpp - calculates the total sales
2  //Created/revised by <your name> on <current date>
3
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9      //declare arrays and variable
10     int company[2][6] = {{12000, 45000, 32000,
11                           67000, 24000, 55000},
12                          {10000, 56000, 42000,
13                           23000, 12000, 34000}};
14     int companySales = 0; //accumulator
15
16     //accumulate sales
17     for (int location = 0; location < 2; location += 1)
18          for (int month = 0; month < 6; month += 1)
19               companySales += company[location][month];
20          //end for
21     //end for
22
23     //display total sales
24     cout << "Company sales: $" << companySales << endl;
25
26     system("pause");
27     return 0;
28 }  //end of main function
```

Figure 12-23 Code for Lab 12-1

QUESTIONS

1. What value is stored in the `company[1][5]` element?
2. How can you calculate the total company sales made in February?
3. What is the highest row subscript in the `company` array? What is the highest column subscript in the array?
4. If the January domestic sales are stored in the `company[0][0]` element, where are the January international sales stored?
5. Change the outer loop's `for` statement to a `while` statement.
6. If you change the `for` clause in Line 18 to `for (int month = 1; month <= 6; month += 1)`, how will the change affect the assignment statement in the `for` loop?
7. Follow the instructions for starting C++ and opening the Lab12-1.cpp file. The file is contained in either the Cpp6\Chap12\Lab12-1 Project folder or the Cpp6\Chap12 folder. Run the program. The total company sales are $412000.
8. Modify the program so that it displays the total domestic sales, total international sales, and total company sales. Save and then run the program.

9. Now, modify the program so that it also displays the total sales made in each month. Save and then run the program.



LAB 12-2 Plan and Create

In this lab, you will plan and create an algorithm for Falcon Incorporated. The problem specification, IPO chart information, and C++ instructions are shown in Figure 12-24. The program displays a shipping charge based on the number of items ordered, which is entered by the user. The problem specification shows the three shipping charges along with their associated minimum and maximum orders. Notice that the maximum order amounts are stored in the first column of the two-dimensional `shipCharges` array, while their corresponding shipping charges are stored in the second column. To find the appropriate shipping charge, you search the first column for the number of items ordered, beginning with the first row. You continue searching the first column in each row as long as there are rows left to search and the number of items ordered is greater than the value in the first column. You stop searching either when there are no more rows to search or when the number of items ordered is less than or equal to the value in the first column. For example, if the number of items ordered is 75, the first value you would look at in the array is 50. The number of items ordered (75) is greater than 50, so you continue searching with the value in the second row (100). The number 100 is greater than the number of items ordered (75), so you stop searching. The appropriate shipping charge is located in the same row—in this case, the second row—but in the second column. The appropriate shipping charge is $10.

Problem specification

Falcon Incorporated wants a program that displays a shipping charge based on the number of items ordered by the customer. The shipping charge information is shown here.

Minimum order	Maximum order	Shipping charge ($)
1	50	20
51	100	10
101	999999	0

IPO chart information	**C++ instructions**
Input	
number ordered	`int numOrdered = 0;`
maximum orders and shipping charges	stored in the array
Processing	
array (3 rows, 2 columns)	`int shipCharges[3][2] = {{50, 20}, {100, 10}, {999999, 0}};`
row subscript counter (0 to 2)	`int rowSub = 0;`

Figure 12-24 Problem specification, IPO chart information, and C++ instructions for Lab 12-2 *(continues)*

(continued)

Output

shipping charge displayed from the array

Algorithm

1. enter the number ordered	`cout << "Number ordered ";`
	`cout << "(negative number`
	`or 0 to end): ";`
	`cin >> numOrdered;`

2. repeat while (the number ordered is greater than 0 and less than or equal to 999999)	`while (numOrdered > 0 &&`
	`numOrdered <= 999999)`
	`{`

assign 0 to the row subscript counter to ensure the search begins in the first row	`rowSub = 0;`

repeat while (the row subscript counter is less than 3 and the number ordered is greater than the value located in the first column of the current row in the array)	`while (rowSub < 3 &&`
	`numOrdered >`
	`shipCharges[rowSub][0])`

add 1 to the row subscript counter to continue the search in the next row	`rowSub += 1;`
end repeat	`//end while`

display the shipping charge	`cout << "Shipping charge`
	`for a quantity of "`
	`<< numOrdered << " is $"`
	`<< shipCharges[rowSub][1]`
	`<< endl << endl;`

enter the number ordered	`cout << "Number ordered ";`
	`cout << "(negative number`
	`or 0 to end): ";`
	`cin >> numOrdered;`

end repeat	`//end while`

Figure 12-24 Problem specification, IPO chart information, and C++ instructions for Lab 12-2

Figure 12-25 shows the code for the entire Falcon Incorporated program, and Figure 12-26 shows the completed desk-check table for the program, assuming the user enters the numbers 75, 200, and -1 as the number of items ordered.

```
 1  //Lab12-2.cpp - displays the shipping charge
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  using namespace std;
 6
 7  int main()
 8  {
 9     //declare array and variables
10     int shipCharges[3][2] = {{50, 20},
11                              {100, 10},
12                              {999999, 0}};
13     int numOrdered = 0;
14     int rowSub     = 0;
15
16     //enter the number ordered
17     cout << "Number ordered ";
18     cout << "(negative number or 0 to end): ";
19     cin >> numOrdered;
20
21     while (numOrdered > 0 && numOrdered <= 999999)
22     {
23          //search array
24          rowSub = 0;
25          while (rowSub < 3 &&
26               numOrdered > shipCharges[rowSub][0])
27                    rowSub += 1;
28          //end while
29
30          //display shipping charge
31          cout << "Shipping charge for a quantity of "
32               << numOrdered << " is $"
33               << shipCharges[rowSub][1] << endl << endl;
34
35          //enter the number ordered
36          cout << "Number ordered ";
37          cout << "(negative number or 0 to end): ";
38          cin >> numOrdered;
39     }    //end while
40
41     system("pause");
42     return 0;
43  }  //end of main function
```

you also can use rowSub++;

your C++ development tool may not require this statement

Figure 12-25 Falcon Incorporated program

shipCharges[0][0]	shipCharges[0][1]
50	20
shipCharges[1][0]	shipCharges[1][1]
100	10
shipCharges[2][0]	shipCharges[2][1]
999999	0

numOrdered	rowSub
0	0
~~75~~	~~0~~
	~~1~~
~~200~~	~~0~~
	~~1~~
	2
-1	

Figure 12-26 Completed desk-check table for the Falcon Incorporated program

The final step in the problem-solving process is to evaluate and modify (if necessary) the program. Recall that you evaluate a program by entering its instructions into the computer and then using the computer to run (execute) it. While the program is running, you enter the same sample data used when desk-checking the program.

DIRECTIONS

Follow the instructions for starting your C++ development tool. Depending on the development tool you are using, you may need to create a new project; if so, name the project Lab12-2 Project and save it in the Cpp6\Chap12 folder. Enter the instructions shown in Figure 12-25 in a source file named Lab12-2.cpp. (Do not enter the line numbers.) Save the file in either the project folder or the Cpp6\Chap12 folder. Now, follow the appropriate instructions for running the Lab12-2.cpp file. Test the program using the same data you used to desk-check the program. If necessary, correct any bugs (errors) in the program.



LAB 12-3 Modify

If necessary, create a new project named Lab12-3 Project. Enter (or copy) the Lab12-2.cpp instructions into a new source file named Lab12-3.cpp. Change Lab12-2.cpp in the first comment to Lab12-3.cpp. Replace the maximum amounts in the `shipCharges` array with the minimum amounts. Then, make the appropriate modifications to the program. Save and then run the program. Test the program appropriately.



LAB 12-4 Desk-Check

Desk-check the Jenko Booksellers program, which is shown in Figure 12-12 in the chapter.



LAB 12-5 Debug

Follow the instructions for starting C++ and opening the Lab12-5.cpp file. The file is contained in either the Cpp6\Chap12\Lab12-5 Project folder or the Cpp6\Chap12 folder. Debug the program.

Summary

- A two-dimensional array resembles a table in that the elements are in rows and columns. Each element has the same data type.
- You can determine the number of elements in a two-dimensional array by multiplying the number of its rows by the number of its columns.
- Each element in a two-dimensional array is identified by a unique combination of two subscripts. The first subscript represents the element's row location in the array, and the second subscript represents its column location. You refer to each element in a two-dimensional array by the array's name and the element's subscripts, which are specified in two sets of square brackets immediately following the name.
- The first row subscript in a two-dimensional array is 0; the first column subscript also is 0. The last row subscript is always one number less than the number of rows in the array. The last column subscript is always one number less than the number of columns in the array.
- You must declare a two-dimensional array before you can use it. When declaring a two-dimensional array, you must provide the number of rows as well as the number of columns.
- After declaring a two-dimensional array, you can use an assignment statement or the extraction operator to enter data into the array.
- You need to use two loops to access every element in a two-dimensional array. One of the loops keeps track of the row subscript. The other loop keeps track of the column subscript.
- To pass a two-dimensional array to a function, you include the array's name in the statement that calls the function. The array's corresponding formal parameter in the function header must specify the formal parameter's data type and name, followed by two sets of square brackets. The first bracket contains the number of rows, and the second bracket contains the number of columns.

Key Term

Two-dimensional array—an array made up of rows and columns; each element has the same data type and is identified by a unique combination of two subscripts: a row subscript and a column subscript

Review Questions

1. The first element in a two-dimensional array has a row subscript of _____ and a column subscript of _____.

   a. 0, 0

   b. 0, 1

   c. 1, 0

   d. 1, 1

2. Which of the following statements creates a two-dimensional `int` array named `rates` that contains three rows and four columns?

   a. `int rates[3, 4] = {0};`

   b. `int rates[4, 3] = {0};`

   c. `int rates[3][4] = {0};`

   d. `int rates[4][3] = {0};`

   Use the `sales` array to answer Review Questions 3 through 6. The array was declared using the `int sales[2][5] = {{10000, 12000, 900, 500, 20000}, {350, 600, 700, 800, 100}};` statement.

3. The statement `sales[1][3] = sales[1][3] + 10;` will replace the number _____.

   a. 900 with 910

   b. 500 with 510

   c. 700 with 710

   d. 800 with 810

4. The statement `sales[0][4] = sales[0][4 - 2];` will replace the number _____.

   a. 20000 with 900

   b. 20000 with 19998

   c. 20000 with 19100

   d. 500 with 12000

5. The statement `cout << sales[0][3] + sales[1][3] << endl;` will _____.

   a. display 1300

   b. display 1600

   c. display `sales[0][3] + sales[1][3]`

   d. result in an error

6. Which of the following `if` clauses verifies that the array subscripts stored in the `row` and `col` variables are valid for the `sales` array?

   a. `if (sales[row][col] >= 0 && sales[row][col] < 5)`

   b. `if (sales[row][col] >= 0 && sales[row][col] <= 5)`

   c. `if (row >= 0 && row < 3 && col >= 0 && col < 6)`

   d. `if (row >= 0 && row <= 1 && col >= 0 && col <= 4)`

Exercises



Pencil and Paper

1. Write the code to declare and initialize a two-dimensional `double` array named `balances` that has four rows and six columns. (The answers to TRY THIS Exercises are located at the end of the chapter.) — TRY THIS

2. Write the code to display the contents of the `balances` array from Pencil and Paper Exercise 1. Use two `for` statements to display the array, row by row. (The answers to TRY THIS Exercises are located at the end of the chapter.) — TRY THIS

3. Rewrite the code from Pencil and Paper Exercise 2 to display the array, column by column. — MODIFY THIS

4. Write the code to store the number 100 in each element in the `balances` array from Pencil and Paper Exercise 1. Use two `for` statements. — INTRODUCTORY

5. Rewrite the code from Pencil and Paper Exercise 4 using two `while` statements. — INTRODUCTORY

6. Rewrite the code from Pencil and Paper Exercise 4 using the `do while` statement in the outer loop and the `while` statement in the nested loop. — INTRODUCTORY

7. Write the statement to assign the C++ keyword `true` to the variable located in the third row, first column of a `bool` array named `answers`. — INTERMEDIATE

INTERMEDIATE

8. Write the code to display the sum of the numbers stored in the following three elements in a two-dimensional `double` array named `sales`: the first row, first column; the second row, third column; and the third row, fourth column.

INTERMEDIATE

9. Write the code to subtract the number 1 from each element in a two-dimensional `int` array named `quantities`. The array has 10 rows and 25 columns. Use two `for` statements.

INTERMEDIATE

10. Rewrite the code from Pencil and Paper Exercise 9 using two `while` statements.

INTERMEDIATE

11. Write the code to find the square root of the number stored in the first row, third column in a two-dimensional `double` array named `mathNumbers`. Display the result on the screen.

ADVANCED

12. Write the code to display the largest number stored in the first column of a two-dimensional `int` array named `orders`. The array has five rows and two columns. Use the `for` statement.

ADVANCED

13. Rewrite the code from Pencil and Paper Exercise 12 using the `while` statement.

SWAT THE BUGS

14. The `numbers` array is a two-dimensional `int` array that contains three rows and five columns. The following statement should call the void `calcTotal` function, passing it the `numbers` array: `calcTotal(numbers[3][5]);`. Correct the statement.



Computer

TRY THIS

15. If necessary, create a new project named TryThis15 Project. Enter the C++ instructions from Figure 12-9 into a source file named TryThis15.cpp. Change the filename in the first comment to TryThis15.cpp. Save and then run the program. Test the program using the data shown in Figure 12-10 in the chapter. Change the `for` loops that enter data into the array to `while` loops. Change the `for` loops that display the array's contents to `do while` loops. Save and then run the program. Test the program using the data shown in Figure 12-10 in the chapter. (The answers to TRY THIS Exercises are located at the end of the chapter.)

TRY THIS

16. If necessary, create a new project named TryThis16 Project. Enter the C++ instructions from Figure 12-22 into a source file named TryThis16.cpp. Change the filename in the first comment to TryThis16.cpp. Save and then run the program. As mentioned in the chapter, when you pass a two-dimensional array to a function, the first set of square brackets in its corresponding formal parameter can be empty. Remove the number 4 from the first formal parameter in the function prototype and function header. The `main` function will now need to pass two actual arguments to the `displayArray`

function: the array and the number of rows (regions) in the array. Make the appropriate modifications to the `displayArray` function prototype, function header, and function call. Also modify the outer loop's `for` clause in the `displayArray` function so it uses the number of rows passed to the function rather than the literal constant 4. (The answers to TRY THIS Exercises are located at the end of the chapter.)

17. If necessary, create a new project named ModifyThis17 Project. Enter the C++ instructions from Figure 12-12 into a source file named ModifyThis17.cpp. Change the filename in the first comment. Save and then run the program. Jenko Booksellers has opened another store. The store's sales of paperback and hardcover books are $650.85 and $246.85, respectively. Add the new sales information to the array, and then modify the program appropriately. Save and then run the program.

MODIFY THIS

18. If necessary, create a new project named ModifyThis18 Project. Enter the C++ instructions from Figure 12-17 into a new source file named ModifyThis18.cpp. Change the filename in the first comment. Save and then run the program. Test the program using the following two pay codes: 6 and 5. Enter -1 to stop the program. Add a new pay code and pay rate to the array. The new pay code is 11, and its corresponding pay rate is $23. Make the appropriate modifications to the code. Save and then run the program. Test the program using the following three pay codes: 6, 5, and 11. Enter -1 to stop the program.

MODIFY THIS

19. Follow the instructions for starting C++ and opening the Introductory19.cpp file. The file is contained in either the Cpp6\Chap12\Introductory19 Project folder or the Cpp6\Chap12 folder. The program should calculate the average of the values stored in the `rates` array. It then should display the average rate on the screen. Display the average with two decimal places. Complete the program using the `for` statement. Save and then run the program.

INTRODUCTORY

20. Follow the instructions for starting C++ and opening the Introductory20.cpp file. The file is contained in either the Cpp6\Chap12\Introductory20 Project folder or the Cpp6\Chap12 folder. The program should display the contents of the two-dimensional array, column by column and also row by row. Complete the program using a `while` statement in the outer loops and a `for` statement in the nested loops. Save and then run the program.

INTRODUCTORY

21. If necessary, create a new project named Intermediate21 Project. Also create a new source file named Intermediate21.cpp. Declare a seven-row, two-column `int` array named `temperatures`. The program should prompt the user to enter the highest and lowest temperatures for seven days. Store the highest temperatures in the first column in the array. Store the lowest temperatures in the second column. The program should display the average high temperature and the average low temperature. Display the average temperatures with one decimal place. Save and then run the program. Test the program using the data shown in Figure 12-27.

INTERMEDIATE

Day	Highest	Lowest
1	95	67
2	98	54
3	86	70
4	99	56
5	83	34
6	75	68
7	80	45

Figure 12-27

INTERMEDIATE

22. In this exercise, you modify the program from Computer Exercise 21. If necessary, create a new project named Intermediate22 Project. Copy the instructions from the Intermediate21.cpp file into a source file named Intermediate22.cpp. Change the filename in the first comment. In addition to displaying the average high temperature and average low temperature, the program also should display the highest temperature stored in the first column in the array and the lowest temperature stored in the second column. Save and then run the program. Test the program using the data shown in Figure 12-27.

ADVANCED

23. Follow the instructions for starting C++ and opening the Advanced23.cpp file. The file is contained in either the Cpp6\Chap12\Advanced23 Project folder or the Cpp6\Chap12 folder. Code the program so that it asks the user for a dollar amount by which each price should be increased. The program then should increase each price in the array's first column by that amount. For example, when the user enters the number 10, the program should increase each element's value by $10. Store the updated prices in the second column of the array. After increasing each price, the program should display the contents of the array, row by row. Save and then run the program. Increase each price by $10.

ADVANCED

24. In this exercise, you code an application that displays the number of times a value appears in a two-dimensional array. Follow the instructions for starting C++ and opening the Advanced24.cpp file. The file is contained in either the Cpp6\Chap12\Advanced24 Project folder or the Cpp6\Chap12 folder. Code the program so that it displays the number of times each of the numbers from 1 through 9 appears in the `numbers` array. (Hint: Use a one-dimensional array of counter variables.) Save and then run the program.

ADVANCED

25. If necessary, create a new project named Advanced25 Project. Also create a new source file named Advanced25.cpp. JM Sales employs 10 salespeople. The sales made by the salespeople during the months of January, February, and March are listed in Figure 12-28. Store the sales amounts in a two-dimensional array. The sales manager wants an application that allows him to enter the current bonus rate. The program should display each salesperson's number (1 through 10), total sales amount, and total bonus amount. It also should display the total bonus paid to all salespeople. Display the bonus amounts with two decimal places. Save and then run the program. Test the program using 10% as the bonus rate.

Salesperson	January	February	March
1	2400	3500	2000
2	1500	7000	1000
3	600	450	2100
4	790	240	500
5	1000	1000	1000
6	6300	7000	8000
7	1300	450	700
8	2700	5500	6000
9	4700	4800	4900
10	1200	1300	400

Figure 12-28

SWAT THE BUGS

26. Follow the instructions for starting C++ and opening the SwatTheBugs26.cpp file. The file is contained in either the Cpp6\Chap12\SwatTheBugs26 Project folder or the Cpp6\Chap12 folder. Debug the program.

Answers to TRY THIS Exercises



Pencil and Paper

1. `double balances[4][6] = {0.0};`

2.
```
for (int row = 0; row < 4; row += 1)
   for (int col = 0; col < 6; col += 1)
      cout << balances[row][col] << endl;
   //end for
//end for
```

you also can use `row++`

you also can use `col++`



Computer

15. See Figure 12-29.

```
 1  //TryThis15.cpp
 2  //Displays the contents of a two-dimensional array
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  using namespace std;
 7
 8  int main()
 9  {
10     //declare and initialize array
11     int orders[4][3] = {0};
12
13     //enter data into the array
14     int region = 0;
15     int month  = 0;
16     while (region < 4)
17     {
18          while (month < 3)
19          {
20               cout << "Number of orders for Region "
21                    << region + 1 << ", Month "
22                    << month + 1 << ": ";
23               cin >> orders[region][month];
24               month += 1;
25          }    //end while
26          region += 1;
27          month = 0;
28     }    //end while
29
30     //display the contents of the array
31     region = 0;
32     month = 0;
33     do   //begin outer loop
34     {
35          cout << "Region " << region + 1
36               << ": " << endl;
37          do   //begin nested loop
38          {
39          cout << " Month " << month + 1
40                    << ": ";
41          cout << orders[region][month] << endl;
42               month += 1;
43          }    while (month < 3);
44          region += 1;
45          month = 0;
46     }    while (region < 4);
47
48     system("pause");
49     return 0;
50 }   //end of main function
```

your C++ development tool may not require this statement



You also can use `month++;` in Lines 24 and 42, and `region++;` in Lines 26 and 44.

Figure 12-29

16. See Figure 12-30. The changes are shaded in the figure.

```
 1  //TryThis16.cpp
 2  //Displays the contents of a two-dimensional array
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  using namespace std;
 7
 8  //function prototype
 9  void displayArray(int nums[][3], int numRegions);
10
11  int main()
12  {
13     //declare and initialize array
14     int orders[4][3] = {0};
15
16     //enter data into the array
17     for (int region = 0; region < 4; region += 1)
18          for (int month = 0; month < 3; month += 1)
19          {
20               cout << "Number of orders for Region "
21                    << region + 1 << ", Month "
22                    << month + 1 << ": ";
23               cin >> orders[region][month];
24          }    //end for
25     //end for
26
27     //display the contents of the array
28     displayArray(orders, 4);
29
30     system("pause");
31     return 0;
32  }  //end of main function
33
34  //*****function definitions*****
35  void displayArray(int nums[][3], int numRegions)
36  {
37     for (int region = 0; region < numRegions; region += 1)
38     {
39          cout << "Region " << region + 1
40               << ": " << endl;
41          for (int month = 0; month < 3; month += 1)
42          {
43               cout << " Month " << month + 1
44                    << ": ";
45               cout << nums[region][month] << endl;
46          }   //end for
47     }    //end for
48  }  //end of displayArray function
```

your C++ development tool may not require this statement



You also can use `region++` in Lines 17 and 37, and `month++` in Lines 18 and 41.

Figure 12-30

CHAPTER 13

Strings

After studying Chapter 13, you should be able to:

- Utilize `string` memory locations in a program
- Get string input using the `getline` function
- Ignore characters using the `ignore` function
- Determine the number of characters in a string
- Access the characters in a string
- Search a string
- Remove characters from a string
- Replace characters in a string
- Insert characters within a string
- Duplicate characters within a string
- Concatenate strings

The string Data Type

In the programs created in the previous chapters, you used memory locations (variables and named constants) having the int and double data types. In this chapter, you will use string memory locations. As you learned in Chapter 3, the string data type is not one of the fundamental data types in C++. Rather, it was added to the C++ language through the use of a class, called the string class. Recall that a class is simply a group of instructions that the computer uses to create an object. The instructions for creating a string object, which can be either a string variable or a string named constant, are contained in the string file. Therefore, for a program to use the string class, it must contain the #include <string> directive. Figure 13-1 shows examples of using the string class to create and initialize string variables and string named constants. Memory locations having the string data type are initialized using string literal constants. Recall from Chapter 3 that a string literal constant is zero or more characters enclosed in double quotation marks. The declaration statement in Example 1 creates a string variable named zipCode and initializes it to the empty string (""), which is the value typically used to initialize string variables. The declaration statement in Example 2 creates a string variable named playAgain and initializes it to the string "Y". Examples 3 and 4 create and initialize string named constants called VALID_LENGTH and COMPANY_NAME.

HOW TO Declare and Initialize string Variables and Named Constants

Example 1

```
string zipCode = "";
```

declares and initializes a string variable named zipCode

Example 2

```
string playAgain = "Y";
```

declares and initializes a string variable named playAgain

Example 3

```
const string VALID_LENGTH = "Valid length";
```

declares and initializes a string named constant called VALID_LENGTH

Example 4

```
const string COMPANY_NAME = "ABC Company";
```

declares and initializes a string named constant called COMPANY_NAME

Figure 13-1 How to declare and initialize string variables and named constants



The amount of memory required to store the strings in each example in Figure 13-1 is different and is managed by the string class.



You learned about the empty string in Chapter 3.

Classes also are referred to as user-defined data types, and they allow programmers to extend the C++ language by defining the characteristics and attributes of a new data type (class). Typically, the characteristics are values, and the attributes are behaviors associated with the class. The behaviors usually are implemented as member functions associated with each class. As a result, each class has its own set of member functions. In this chapter, you will explore a few of the more commonly used member functions associated with the string class.

The Creative Sales Program

Figure 13-2 shows the problem specification and IPO chart for the Creative Sales program, which gets a salesperson's name and sales amount from the keyboard. It then calculates the salesperson's bonus and displays the salesperson's name and bonus amount on the screen.

Problem specification
Creative Sales wants a program that allows the sales manager to enter a salesperson's name and sales amount. The program should calculate the salesperson's bonus by multiplying the sales amount by 10%. It then should display the salesperson's name and bonus amount on the screen.

Input	Processing	Output
name sales rate (10%)	Processing items: none Algorithm: 1. enter the name and sales 2. calculate the bonus by multiplying the sales by the rate 3. display the name and bonus	name bonus

Figure 13-2 Problem specification and IPO chart for the Creative Sales program

So far, you have used the extraction operator (>>) to get numbers and characters from the user at the keyboard. The extraction operator also can be used to get string input from the keyboard, as shown in the examples in Figure 13-3. (For clarity, the variable declaration statements are included in the examples.) However, keep in mind that the extraction operator stops reading characters when it encounters a white-space character in the input. A white-space character is a blank, tab, or newline. You enter a blank character when you press the Spacebar on your keyboard. You enter a tab character when you press the Tab key, and you enter a newline character when you press the Enter key. As a result, if the user inadvertently enters the string "45 602" (rather than "45602") as the ZIP code in Example 1, the extraction operator in the `cin >> zipCode;` statement will store only the string "45" in the `zipCode` variable.



You learned about the extraction operator in Chapter 4.

HOW TO Use the Extraction Operator (>>) to Get String Input

Example 1

```
string zipCode = "";
cout << "Enter your zip code: ";
cin >> zipCode;
```

gets a string from the keyboard and stores it in the `zipCode` variable

Example 2

```
string playAgain = "Y";
cout << "Play the game again? (Y/N): ";
cin >> playAgain;
```

gets a string from the keyboard and stores it in the `playAgain` variable

Figure 13-3 How to use the extraction operator (>>) to get string input

Because many strings entered at the keyboard contain one or more blank characters (such as "Bowling Green, Kentucky" and "Gerald R. Jones"), the `string` class provides a member function for accepting that type of input. The function is called `getline`. You will need to use the `getline` function to get the salesperson's name in the Creative Sales program.

The `getline` Function

Figure 13-4 shows the syntax for using the `getline` function to get string input from the keyboard. The semicolon that appears as the last character in the syntax indicates that the `getline` function is a self-contained statement. The function has three actual arguments, two of which are required. The required `cin` argument refers to the computer keyboard, and the required *stringVariableName* argument is the name of a `string` variable in which to store the input. You can use the optional *delimeterCharacter* argument to indicate the end of the string. The argument represents the character that immediately follows the last character in the string. The **getline function** will continue to read the characters entered at the keyboard until it encounters the delimiter character. If you omit the *delimiterCharacter* argument, the default delimiter character is the newline character. For example, if the user types the words "Good night" and then presses the Enter key, the string will end with the letter t, which is the last character the user typed before pressing the Enter key. When the `getline` function encounters the delimiter character in the input, it discards the character—a process C++ programmers refer to as **consuming the character**. Also included in Figure 13-4 are examples of using the `getline` function. The statement in Example 1 tells the computer to read the characters entered at the keyboard and then store the characters in the `name` variable. The computer will stop reading and storing characters when it encounters the newline character, which is when the user presses the Enter key. (Recall that the default delimiter character in the `getline` function is the newline character.) At that point, the computer will consume (discard) the newline character. The statement in Example 2 also tells the computer to read the characters entered at the keyboard and store them in the `name` variable. Here again, the computer will stop reading and storing characters when it encounters the newline character, which will be consumed by the computer. The newline character is designated in C++ by a backslash and the letter n, both enclosed in single quotation marks, like this: `'\n'`. Although the newline character consists of two characters, it is treated as one character by the computer. The backslash in the newline character is called an escape character, and it indicates that the character that follows it—in this case, the letter n—has a special meaning. The combination of the backslash and the character that follows it is called an escape sequence. The statement in Example 3 in Figure 13-4 tells the computer to read the characters entered at the keyboard and store them in the `city` variable. In this case, the computer will stop reading and storing characters when it encounters the # character, which will be consumed.



An example of another escape sequence is `'\t'`, which represents the Tab key.

HOW TO Use the `getline` Function to Get String Input from the Keyboard

Syntax

getline(cin, *stringVariableName*[**,** *delimeterCharacter*]**);**

Example 1

```
string name = "";
cout << "Enter your name: ";
getline(cin, name);
```

stores the characters entered by the user, up until the newline character, in the `name` variable; consumes the newline character

Example 2

```
string name = "";
cout << "Enter your name: ";
getline(cin, name, '\n');
```

same as Example 1, but specifies the newline delimiter character

Example 3

```
string city = "";
cout << "City: ";
getline(cin, city, '#');
```

stores the characters entered by the user, up until the # character, in the `city` variable; consumes the # character

Figure 13-4 How to use the `getline` function to get string input from the keyboard



Recall that the newline character, which represents the Enter key, is the default delimiter character in the `getline` function.

Figure 13-5 shows the Creative Sales program, with the code pertaining to `string` data shaded. The `#include <string>` directive, which is necessary when using `string` memory locations, appears on Line 7. The declaration statement on Line 13 creates a `string` variable called `name` and initializes it to the empty string. The `getline` function on Line 19 waits for the user to respond to the "Salesperson's name: " prompt. When the user presses the Enter key, the function stores the characters typed by the user, up until the newline character, in the `name` variable; it then consumes the newline character. Figure 13-6 shows a sample run of the program.

```
1  //Creative Sales.cpp
2  //Displays a salesperson's name and bonus
3  //Created/revised by <your name> on <current date>
4
5  #include <iostream>
6  #include <iomanip>
7  #include <string>
8  using namespace std;
9
```

Figure 13-5 Creative Sales program *(continues)*

(continued)

```
10  int main()
11  {
12      const double RATE = .1;
13      string name       = "";
14      int sales         = 0;
15      double bonus      = 0.0;
16
17      //get input
18      cout << "Salesperson's name: ";
19      getline(cin, name);
20      cout << "Sales: ";
21      cin >> sales;
22
23      //calculate bonus
24      bonus = sales * RATE;
25
26      //display output
27      cout << fixed << setprecision(2);
28      cout << "Bonus for " << name
29           << ": $" << bonus << endl;
30
31      system("pause");
32      return 0;
33  }   //end of main function
```

your C++ development tool may not require this statement

Figure 13-5 Creative Sales program

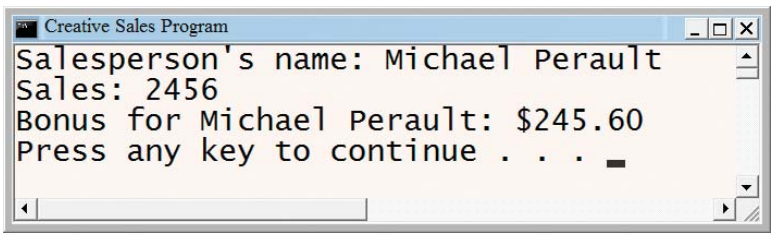

Creative Sales Program
Salesperson's name: Michael Perault
Sales: 2456
Bonus for Michael Perault: $245.60
Press any key to continue . . .

Figure 13-6 Sample run of the Creative Sales program

Now let's make a slight change to the problem specification for Creative Sales. In addition to entering the salesperson's name and sales amount, the sales manager also should enter the state in which the sales were made. Consider how this change will affect the original program shown in Figure 13-5. The modified program will need to declare and initialize a `string` variable to store the state name entered by the user. It also will need both a `cout` statement that prompts the user to enter the state name and a `getline` function to get the user's input. A `getline` function is appropriate because some state names—for example, North Carolina—contain a blank character. The modifications to the original program are shaded in the partial program shown in Figure 13-7. Figure 13-8 shows a sample run of the modified program. Notice that the program does not pause to allow the user to enter the state name.

```
12      const double RATE = .1;
13      string name       = "";
14      int sales         = 0;
15      double bonus      = 0.0;
16      string state      = "";
17
18      //get input
19      cout << "Salesperson's name: ";
20      getline(cin, name);
21      cout << "Sales: ";
22      cin >> sales;
23      cout << "State name: ";
24      getline(cin, state);
25
26      //calculate bonus
27      bonus = sales * RATE;
```

Figure 13-7 Partial Creative Sales program showing the modifications

the user was not given a chance to enter the state name

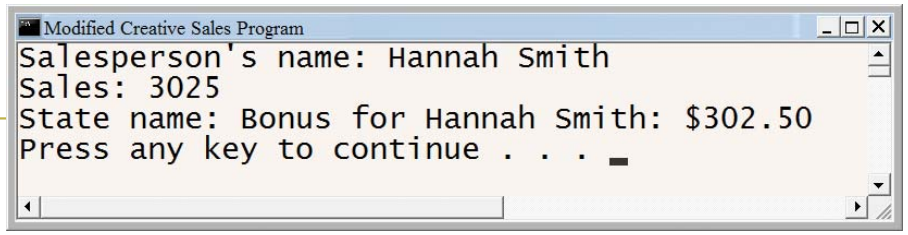

Modified Creative Sales Program
Salesperson's name: Hannah Smith
Sales: 3025
State name: Bonus for Hannah Smith: $302.50
Press any key to continue . . .

Figure 13-8 Result of running the modified Creative Sales program

To understand why the modified program is not working correctly, you need to understand how the extraction operator and `getline` function get keyboard input. Toward this end, you will desk-check Lines 19 through 24 in the partial program shown in Figure 13-7. You will use Hannah Smith, 3025, and Alaska as the salesperson's name, sales amount, and state. The `cout` statement on Line 19 prompts the user to enter the salesperson's name. Before allowing the user to enter the name, the `getline` function on Line 20 checks the `cin` object to determine whether it contains any characters. As you learned in Chapter 4, the `cin` object stores the characters entered at the keyboard. Because the `cin` object is empty at this point in the program, the `getline` function waits for the user to enter a name. In this case, the user types the string "Hannah Smith" and then presses the Enter key to indicate that he or she is finished entering the name. The computer stores the string and the newline character (`'\n'`) in the `cin` object. It then alerts the `getline` function that the object now contains data. The `getline` function removes both the string and the newline character from the `cin` object. It stores the string in the `name` variable and then consumes the newline character. Next, the `cout` statement on Line 21 prompts the user to enter the sales. Before allowing the user to enter the sales, the extraction operator in the `cin >> sales;` statement on Line 22 checks the `cin` object to determine

whether it contains any characters. The object is empty at this point, so the extraction operator waits for the user to enter a sales amount. In this case, the user types the four numbers 3, 0, 2, and 5 and then presses the Enter key to indicate that he or she is finished entering the sales amount. The computer stores the four numbers and the newline character (`'\n'`) in the `cin` object. It then alerts the extraction operator that the object now contains data. The extraction operator removes the four numbers from the `cin` object and stores them in the `sales` variable. However, it leaves the newline character in the object. The statement on Line 23 prompts the user to enter the state name. Before allowing the user to respond to the prompt, the `getline` function on Line 24 checks the `cin` object to determine whether it contains any characters. At this point, the object contains the newline character, which the `getline` function interprets as the end of the state name entry. As a result, the `getline` function stores the empty string in the `state` variable and then consumes the newline character. Processing continues with the bonus calculation statement on Line 27. Obviously, the program is not working correctly because of the newline character that the extraction operator on Line 22 leaves in the `cin` object. You can fix the program by telling the computer to ignore that character.

The `ignore` Function

You can use the C++ **ignore function** to instruct the computer to first read and then ignore characters stored in the `cin` object. The function ignores the characters by consuming (discarding) them. Figure 13-9 shows the function's syntax. Like the `getline` function, the `ignore` function is a self-contained statement, as the semicolon at the end of the syntax indicates. The function has two actual arguments, both of which are optional. The *numberOfCharacters* argument is an integer that represents the maximum number of characters the function should consume. If you omit the *numberOfCharacters* argument, the default number of characters to consume is 1. The *delimiterCharacter* argument is a character that, when consumed, stops the `ignore` function from reading and discarding any additional characters. The `ignore` function stops reading and discarding characters either when it consumes the number of characters specified in the *numberOfCharacters* argument or when it consumes the *delimiterCharacter*, whichever occurs first. Figure 13-9 also shows examples of using the `ignore` function. As indicated in Example 1, you can use either the statement `cin.ignore();` or the statement `cin.ignore(1);` to have the computer read and then discard (consume) one character. The statement in Example 2 tells the computer to read and consume five characters. The statement in Example 3 tells the computer to read and consume characters until either 100 characters are consumed or the newline character is consumed, whichever occurs first. When the computer processes the statement in Example 4, it will read and discard characters until either 25 characters are consumed or the # character is consumed, whichever occurs first.

HOW TO Use the `ignore` Function

Syntax

cin.ignore([*numberOfCharacters*] [**,** *delimiterCharacter*]**);**

Example 1

```
cin.ignore();
```

reads and consumes one character; also can be written as `cin.ignore(1);`

Example 2

```
cin.ignore(5);
```

reads and consumes five characters

Example 3

```
cin.ignore(100, '\n');
```

reads and consumes characters until either 100 characters are consumed or the newline character is consumed, whichever occurs first

Example 4

```
cin.ignore(25, '#');
```

reads and consumes characters until either 25 characters are consumed or the # character is consumed, whichever occurs first

Figure 13-9 How to use the `ignore` function



Lab 13-1 shows another example of a program that requires the `ignore` function.

In the modified Creative Sales program, you will use the `ignore` function to consume the newline character that is left in the `cin` object after the sales amount is entered. You do this by entering the `ignore` function immediately after the `cin` statement that gets the sales amount. Figure 13-10 shows the modified program with the `ignore` function entered on Line 23. The modifications made to the original program (shown earlier in Figure 13-5) are shaded in Figure 13-10. Figure 13-11 shows a sample run of the modified program.

```
 1  //Modified Creative Sales.cpp
 2  //Displays a salesperson's name and bonus
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  #include <iomanip>
 7  #include <string>
 8  using namespace std;
 9
10  int main()
11  {
12      const double RATE = .1;
13      string name       = "";
14      int sales         = 0;
15      double bonus      = 0.0;
16      string state      = "";
```

Figure 13-10 Modified Creative Sales program with the `ignore` function *(continues)*

(continued)

```
17
18      //get input
19      cout << "Salesperson's name: ";
20      getline(cin, name);
21      cout << "Sales: ";
22      cin >> sales;
23      cin.ignore(100, '\n');
24      cout << "State: ";
25      getline(cin, state);
26
27      //calculate bonus
28      bonus = sales * RATE;
29
30      //display output
31      cout << fixed << setprecision(2);
32      cout << "Bonus for " << name
33           << ": $" << bonus << endl;
34
35      system("pause");
36      return 0;
37  }   //end of main function
```

your C++ development tool may not require this statement

Figure 13-10 Modified Creative Sales program with the `ignore` function

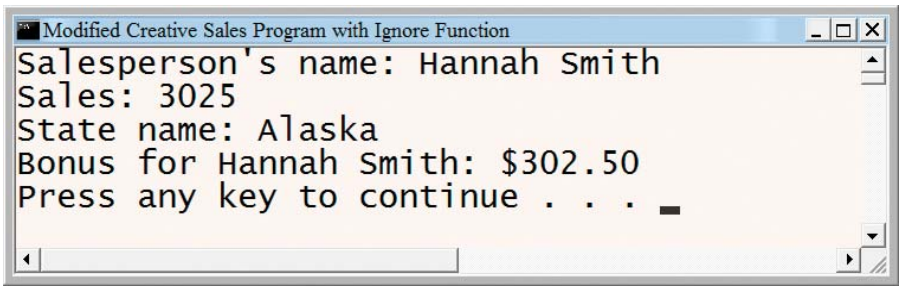

Modified Creative Sales Program with Ignore Function
Salesperson's name: Hannah Smith
Sales: 3025
State name: Alaska
Bonus for Hannah Smith: $302.50
Press any key to continue . . .

Figure 13-11 Sample run of the modified Creative Sales program with the `ignore` function

You will need to use the `ignore` function whenever the `getline` function is processed after a statement containing the extraction operator.

You may be wondering why the modified program in Figure 13-10 uses the `cin.ignore(100, '\n');` statement rather than the simpler `cin.ignore();` statement. Although both statements will consume the newline character left in the `cin` object after the sales amount is entered, there is an advantage to using the `cin.ignore(100, '\n');` statement in the program. To illustrate, assume that when entering the sales amount, the user types the four numbers 3, 0, 2, and 5, followed inadvertently by the letter L, and then presses the Enter key. The computer stores the four numbers, along with the letter L and the newline character, in the `cin` object. It then alerts the extraction operator in the `cin >> sales;` statement that the object now contains data. The extraction operator removes the four numbers from the `cin` object and stores them in the `sales` variable. However, it leaves both the letter L (which cannot be stored in an `int` variable) and the newline character in the object; at this point, the `cin` object contains two characters. If the program used the `cin.ignore();` statement, the `ignore`

function would consume only the letter L. The newline character would still be in the `cin` object when the `getline(cin, state);` statement is processed. As you learned earlier, the `getline` function will interpret the newline character as the end of the state name entry. The `cin.ignore(100, '\n');` statement, on the other hand, will consume both the letter L and the newline character. This is because the statement tells the computer to read and discard characters until either 100 characters are consumed or the newline character is consumed, whichever occurs first. As a result, the `getline` function will not find any characters in the `cin` object and will wait for the user to enter the state name.



The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 13-1

1. Which of the following declares a `string` named constant and initializes it to the name of the first U.S. President?

   a. `const String FIRST_PRES = "George Washington";`

   b. `const string FIRST_PRES = 'George Washington';`

   c. `const string FIRST_PRES = "George Washington";`

   d. `constant string FIRST_PRES = "George Washington";`

2. Which of the following declares a `string` variable named `country` and initializes it to the empty string?

   a. `String country = "";`

   b. `string country = "";`

   c. `string country = ' ';`

   d. `String country = '';`

3. Which of the following gets a string of characters from the `cin` object and stores them in a `string` variable named `streetAddress`?

   a. `getline(cin, streetAddress, '\n');`

   b. `getline(streetAddress, cin);`

   c. `cin.getline(streetAddress);`

   d. `getline.cin(streetAddress);`

4. Which of the following tells the computer to stop reading and discarding characters either when it consumes 10 characters or when the user presses the Enter key, whichever occurs first?

   a. `cin.ignore('\n', 10);`

   b. `cin.ignore(10);`

   c. `cin.ignore(10, '\n');`

   d. both b and c

The ZIP Code Program

Many times, a program will need to manipulate (process) string data in some way. For example, it may need to look at the first character in an inventory part number to determine the part's location in the warehouse. Or, it may need to search an address to determine the street name. Or, as in the ZIP code program created in this section, it may need to determine the number of characters in a ZIP code. The problem specification and IPO chart for the ZIP code program are shown in Figure 13-12. As the problem specification states, the program should get a ZIP code from the user and then verify that the user's entry contains exactly five characters. If the ZIP code contains the required five characters, the program should display the message "Valid length"; otherwise, it should display the message "Invalid length". The program will store the ZIP code in a variable, because its value will change as the program is running. It will store the two messages, on the other hand, in named constants, because their values will not change during runtime. Because the ZIP code and messages will not be used in any calculations, the program will declare the variable and named constants using the `string` data type.

Problem specification
Create a program that allows the user to enter a ZIP code. The program should verify that the user entered exactly five characters. If the user entered the required number of characters, the program should display the message "Valid length"; otherwise, it should display the message "Invalid length". The ZIP code will not be used in a calculation, so the program should store its value in a `string` variable. The program should store the two messages in two `string` named constants.

Input	Processing	Output
ZIP code	Processing items: none	"Valid length" message or "Invalid length" message

```
Algorithm:
1. enter the ZIP code
2. repeat while (the ZIP code is not "-1")
       if (the ZIP code contains exactly 5 characters)
          display the "Valid length" message
       else
          display the "Invalid length" message
       end if
       enter the ZIP code
   end repeat
```

Figure 13-12 Problem specification and IPO chart for the ZIP Code program

Determining the Number of Characters Contained in a `string` Variable

You can use the `string` class's **length function** to determine the number of characters contained in a `string` variable. The function's syntax is shown in Figure 13-13. In the syntax, *string* is the name of the `string` variable whose length you want to determine. The `length` function returns the number of characters contained in the variable. Figure 13-13 also shows examples

of using the `length` function. The `cout` statement in Example 1 uses the `length` function to display the number of characters (15) contained in the `name` variable. The `if` clause in Example 2 uses the `length` function to compare the number of characters stored in the `zipCode` variable with the number 5. The `while` clause in Example 3 uses the `length` function to repeat the loop instructions as long as the `partNum` variable does not contain exactly six characters.

HOW TO Use the `length` Function

Syntax

string.**length()**

Example 1

```
string name = "Nancy Haberdeen";
cout << name.length() << endl;
```

displays the number 15 on the screen

Example 2

```
const string VALID_MSG = "Valid length";
const string INVALID_MSG = "Invalid length";
string zipCode = "";
cout << "Five-character ZIP code: ";
cin >> zipCode;
if (zipCode.length() == 5)
     cout << VALID_MSG << endl;
else
     cout << INVALID_MSG << endl;
//end if
```

compares the number of characters stored in the `zipCode` variable with the number 5 and then displays an appropriate message

Example 3

```
string partNum = "";
cout << "Six-character part number: ";
getline(cin, partNum);
while (partNum.length() != 6)
{
     cout << "Six-character part number: ";
     getline(cin, partNum);
}    //end while
```

continues getting a part number until the user enters exactly six characters

Figure 13-13 How to use the `length` function

Figure 13-14 shows the ZIP code program. The `length` function appears in the `if` clause on Line 21 and is shaded in the figure. Figure 13-15 shows a sample run of the program.

```
 1  //ZIP Code.cpp
 2  //Displays a message indicating whether a ZIP
 3  //code's length is valid or invalid
 4  //Created/revised by <your name> on <current date>
 5
 6  #include <iostream>
 7  #include <string>
 8  using namespace std;
 9
10  int main()
11  {
12      const string VALID_MSG   = "Valid length";
13      const string INVALID_MSG = "Invalid length";
14      string zipCode           = "";
15
16      cout << "Five-character ZIP code (-1 to end): ";
17      cin >> zipCode;
18
19      while (zipCode != "-1")
20      {
21          if (zipCode.length() == 5)
22               cout << VALID_MSG << endl << endl;
23          else
24               cout << INVALID_MSG << endl << endl;
25          //end if
26
27          cout << "Five-character ZIP code (-1 to end): ";
28          cin >> zipCode;
29      }   //end while
30
31      system("pause");
32      return 0;
33  }   //end of main function
```

your C++ development tool may not require this statement

Figure 13-14 ZIP code program

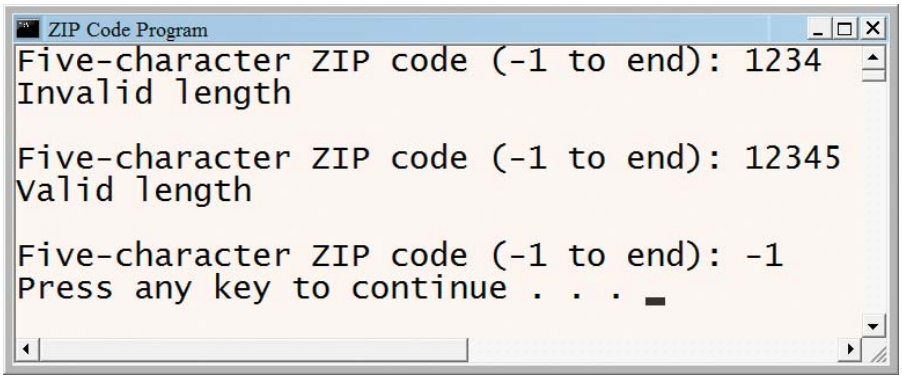

ZIP Code Program
Five-character ZIP code (-1 to end): 1234
Invalid length
Five-character ZIP code (-1 to end): 12345
Valid length
Five-character ZIP code (-1 to end): -1
Press any key to continue . . .

Figure 13-15 Sample run of the ZIP code program

Accessing the Characters Contained in a `string` Variable

In this section, you will modify the ZIP code program shown earlier in Figure 13-14. The modified program will verify that each character entered by the user is a number. You can accomplish this task using a loop along with the `string` class's `length` and `substr` functions. (The "substr" stands for "substring.") The **substr function** allows you to access any number of characters contained in a `string` variable; it then returns the characters. In the function's syntax, which is shown in Figure 13-16, *string* is the name of the `string` variable that contains the characters you want to access. The `substr` function has two actual arguments: *subscript* and *count*. The arguments can be numeric literal constants or the names of numeric variables. The required *subscript* argument represents the subscript of the first character you want to access in the *string*. You learned about subscripts in Chapters 11 and 12, which covered one-dimensional and two-dimensional arrays. A string is equivalent to a one-dimensional array of characters. Each character has a unique subscript that can be used to refer to the contents of the string, one character at a time. The subscript indicates the character's position in the string. The first character in a string has a subscript of 0, the second has a subscript of 1, and so on. To have the `substr` function access the first four characters in a string, you use 0 as the *subscript* argument and 4 as the *count* argument. Similarly, to have the function access the tenth through the twelfth characters, you use 9 as the *subscript* argument and 3 as the *count* argument. The `substr` function returns a string that contains *count* number of characters, beginning with the character whose subscript is specified in the *subscript* argument. If you omit the *count* argument, the `substr` function returns all of the characters from the *subscript* position through the end of the *string*. Also included in Figure 13-16 are examples of using the `substr` function. In Example 1, the `first = name.substr(0, 4);` statement assigns the first four characters contained in the `name` variable ("Jack") to the `first` variable. The `last = name.substr(5);` statement assigns all of the characters contained in the `name` variable, beginning with the character whose subscript is 5, to the `last` variable. In this case, the statement assigns "Blackfeather" to the `last` variable. The condition in the `if` clause in Example 2 uses the `substr` function to compare the first character contained in the `sales` variable with the dollar sign. If the condition evaluates to true, the `sales = sales.substr(1);` statement assigns all of the characters from the `sales` variable, beginning with the character whose subscript is 1, to the `sales` variable. In other words, the statement assigns all of the characters except the dollar sign to the variable. The condition in the `if` clause in Example 3 uses the `substr` and `length` functions to determine whether the string stored in the `rate` variable ends with the percent sign. If it does, the `rate = rate.substr(0, rate.length() - 1);` statement assigns the `rate` variable's contents, excluding the last character (which is the percent sign), to the `rate` variable.

HOW TO Use the `substr` Function

Syntax

string.**substr(***subscript*[**,** *count*]**)**

Example 1

```
string name = "Jack Blackfeather";
string first = "";
string last = "";
first = name.substr(0, 4);
last = name.substr(5);
```

assigns "Jack" to the `first` variable and "Blackfeather" to the `last` variable

Example 2

```
string sales = "";
cout << "Enter the sales: ";
getline(cin, sales);
if (sales.substr(0, 1) == "$")
     sales = sales.substr(1);
//end if
```

if the string stored in the `sales` variable begins with the dollar sign, the code assigns the variable's contents, excluding the dollar sign, to the variable

Example 3

```
string rate = "";
cout << "Enter the rate: ";
getline(cin, rate);
if (rate.substr(rate.length() - 1, 1) == "%")
     rate = rate.substr(0, rate.length() - 1);
//end if
```

if the string stored in the `rate` variable ends with the percent sign, the code assigns the variable's contents, excluding the percent sign, to the variable

Figure 13-16 How to use the `substr` function

Figure 13-17 shows the problem specification, IPO chart information, and C++ code for the modified ZIP code program. The modified program contains a nested loop that determines whether each character entered by the user is a number. The nested loop is shaded in the figure. The condition in the nested loop's `while` clause tells the computer to repeat the loop instructions as long as both of the following two sub-conditions evaluate to true. First, the contents of the `sub` variable, which keeps track of the character subscripts in the `zipCode` variable, must be less than the length of the string stored in the `zipCode` variable. Second, the `foundNonNumber` variable, which keeps track of whether a non-numeric character appears in the `zipCode` variable, must contain the character 'N'. Notice that the condition in the `if` clause inside the nested loop uses the `substr` function and `sub` variable to access the current character in the `zipCode` variable. The condition determines whether the current character is less than "0" or greater than "9". If the condition evaluates to true, it means that the character is not

a number. In that case, the statement in the `if` statement's true path assigns the character 'Y' to the `foundNonNumber` variable. Otherwise, the statement in its false path adds the number 1 to the character subscript, which is stored in the `sub` variable; doing this allows the loop to look at the next character in the `zipCode` variable. The `if` statement that follows the nested loop compares the value stored in the `foundNonNumber` variable to the character 'N'. If the variable contains the character 'N', the `if` statement's true path displays the "All numbers" message; otherwise, its false path displays the "Not all numbers" message.

Problem specification
Create a program that allows the user to enter a ZIP code. The program should verify that the user entered exactly five characters. If the user entered the required number of characters, the program should display the message "Valid length". It then should verify that each of the five characters is a number. If all five characters are numbers, the program should display the message "All numbers"; otherwise, it should display the message "Not all numbers". If, on the other hand, the user did not enter the required number of characters, the program should display the message "Invalid length". The ZIP code will not be used in a calculation, so the program should store its value in a `string` variable. The program should store the four messages in four `string` named constants.

IPO chart information	**C++ instructions**
Input	
ZIP code	`string zipCode = "";`
Processing	
variable that keeps track of whether a non-numeric character appears in the ZIP code ('N')	`char foundNonNumber = 'N';`
character subscript	created and initialized in the if statement's true path
Output	
"Valid length" message or	`const string VALID_MSG = "Valid length";`
"Invalid length" message	`const string INVALID_MSG = "Invalid length";`
"All numbers" message or	`const string ALL_NUMBERS = "All numbers";`
"Not all numbers" message	`const string NOT_ALL_NUMBERS = "Not all numbers";`
Algorithm	
1. enter the ZIP code	`cout << "Five-character ZIP code (-1 to end): "; cin >> zipCode;`

Figure 13-17 Modified problem specification, IPO chart information, and C++ instructions for the ZIP code problem *(continues)*

(continued)

```
2. repeat while (the ZIP code is not "-1")
      if (the ZIP code contains
      exactly 5 characters)
         display the "Valid length" message
         assign 0 to the character subscript
         repeat while (the character subscript
         is less than the number of characters
         in the ZIP code, and the variable
         that keeps track of whether a
         non-numeric character appears
         in the ZIP code contains 'N')
            if (the current character
            in the ZIP code is less than
            "0" or greater than "9")
               assign 'Y' to the variable
               that keeps track of whether
               a non-numeric character
               appears in the ZIP code
            else
               add 1 to the character
               subscript
            end if
         end repeat
         if (the variable that keeps track of
         whether a non-numeric character
         appears in the ZIP code contains 'N')
            display the "All numbers"
            message
         else
            display the "Not all
            numbers" message
         end if
      else
         display the "Invalid length"
         message
      end if
      enter the ZIP code
   end repeat
```

```
while (zipCode != "-1")
{
   if (zipCode.length() == 5)
   {
      cout << VALID_MSG << endl;
      int sub = 0;
      while (sub < zipCode.length()
      && foundNonNumber == 'N')
         if (zipCode.substr(sub, 1)
         < "0" || zipCode.substr
         (sub, 1) > "9")
            foundNonNumber = 'Y';
         else
            sub += 1;
         //end if
      //end while
      if (foundNonNumber == 'N')
         cout << ALL_NUMBERS
         << endl << endl;
      else
         cout << NOT_ALL_NUMBERS
         << endl << endl;
      //end if
   }
   else
      cout << INVALID_MSG
      << endl << endl;
   //end if
   cout << "Five-character ZIP code
   (-1 to end): ";
   cin >> zipCode;
}  //end while
```

Figure 13-17 Modified problem specification, IPO chart information, and C++ instructions for the ZIP code problem

The modified ZIP code program is shown in Figure 13-18, with the modifications made to the original program (shown earlier in Figure 13-14) shaded. Figure 13-19 shows a sample run of the modified program.

```
1  //Modified ZIP Code.cpp
2  //Displays a message indicating whether a ZIP
3  //code's length is valid or invalid. Also displays
4  //a message indicating whether each character
5  //in the ZIP code is a number.
6  //Created/revised by <your name> on <current date>
7
8  #include <iostream>
9  #include <string>
10 using namespace std;
11
12 int main()
13 {
14     const string VALID_MSG       = "Valid length";
15     const string INVALID_MSG     = "Invalid length";
16     const string ALL_NUMBERS     = "All numbers";
17     const string NOT_ALL_NUMBERS = "Not all numbers";
18     string zipCode               = "";
19     char foundNonNumber          = 'N';
20
21     cout << "Five-character ZIP code (-1 to end): ";
22     cin >> zipCode;
23
24     while (zipCode != "-1")
25     {
26         if (zipCode.length() == 5)
27         {
28              cout << VALID_MSG << endl;
29              //determine whether each character
30              //is a number
31              int sub = 0;
32              while (sub < zipCode.length()
33                   && foundNonNumber == 'N')
34                   if (zipCode.substr(sub, 1) < "0"
35                        || zipCode.substr(sub, 1) > "9")
36                             foundNonNumber = 'Y';
37                   else
38                        sub += 1; //check the next character
39                   //end if
40              //end while
41
42              //display message regarding numbers
43              if (foundNonNumber == 'N')
44                   cout << ALL_NUMBERS << endl << endl;
45              else
46                   cout << NOT_ALL_NUMBERS << endl << endl;
47              //end if
48         }
49         else
50              cout << INVALID_MSG << endl << endl;
51         //end if
52
53         cout << "Five-character ZIP code (-1 to end): ";
54         cin >> zipCode;
55     }   //end while
56
57     system("pause");
58     return 0;
59 }   //end of main function
```

your C++ development tool may not require this statement



You also can use `sub++;` in Line 38.

Figure 13-18 Modified ZIP code program

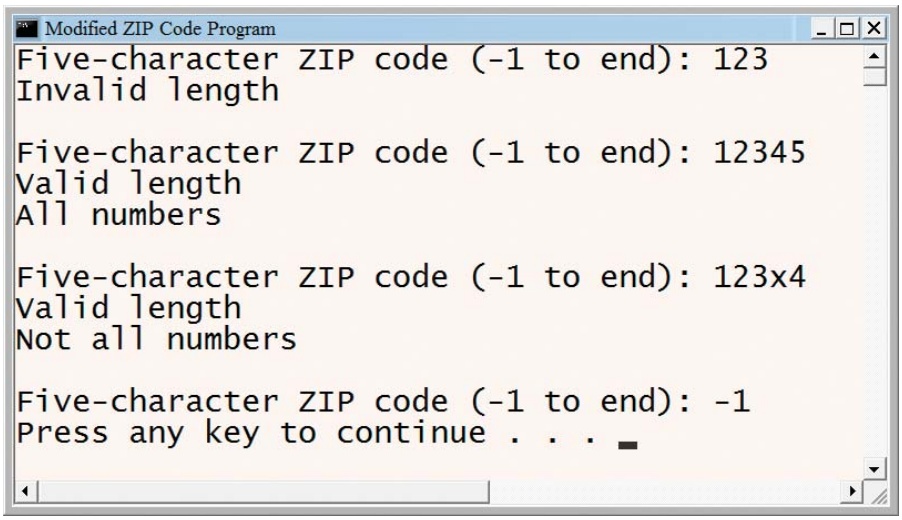

Modified ZIP Code Program
Five-character ZIP code (-1 to end): 123
Invalid length
Five-character ZIP code (-1 to end): 12345
Valid length
All numbers
Five-character ZIP code (-1 to end): 123x4
Valid length
Not all numbers
Five-character ZIP code (-1 to end): -1
Press any key to continue . . .

Figure 13-19 Sample run of the modified ZIP code program

Mini-Quiz 13-2



The answers to Mini-Quiz questions are located in Appendix A.

1. Which of the following `while` clauses tells the computer to process the loop instructions as long as the `employee` variable contains more than 20 characters?

   a. `while (employee.length() > 20)`

   b. `while (employee.length() > "20");`

   c. `while (employee.length() > '20');`

   d. `while (employee.length() > 20);`

2. Write a C++ `if` clause that determines whether a `string` variable named `code` contains seven characters.

3. The `cityState` variable contains the string "Los Angeles, CA". Which of the following assigns the state ID ("CA") to a `string` variable named `state`?

   a. `state = cityState.substr(13);`

   b. `state = cityState.substr(13, 2);`

   c. `state = cityState.substr(14, 2);`

   d. both a and b

4. Write a `cout` statement that displays the last character contained in the `college` variable. The variable has the `string` data type.

The Rearranged Name Program

Figure 13-20 shows the problem specification and IPO chart for the rearranged name program. The program should allow the user to enter a person's first and last names, separated by a space. The program should display the person's last name followed by a comma, a space, and the person's first name. For example, if the user enters "Henry Smith", the program should display "Smith, Henry". As the algorithm indicates, you can accomplish this task by searching for the space character that separates the first name from the last name. The characters to the left of the space character represent the first name. The characters to the right of the space character represent the last name. You will learn how to search the contents of a `string` variable in the next section.

Problem specification
Create a program that allows the user to enter a person's first and last names, separated by a space. The program should display the person's last name followed by a comma, a space, and the person's first name. For example, if the user enters "Henry Smith", the program should display "Smith, Henry".

Input	Processing	Output
name (first name followed by a space and the last name)	Processing items: space's location Algorithm: 1. enter the name 2. search the name, looking for the space's location 3. assign the characters to the left of the space's location as the first name 4. assign the characters to the right of the space's location as the last name 5. display the rearranged name, which is the last name followed by a comma, a space, and the first name	rearranged name (last name followed by a comma, a space, and the first name)

Figure 13-20 Problem specification and IPO chart for the rearranged name program

Searching the Contents of a `string` Variable

You can use the `string` class's **find function** to search the contents of a `string` variable to determine whether it contains a specific sequence of characters. For example, you can use the function to determine whether a phone number contains a certain area code. Or, you can use the function to determine whether a specific street name appears in an address. You also can use it to determine the location of the space character that separates a first name from a last name. Figure 13-21 shows the syntax of the **find** function. In the syntax, *string* is the name of the `string` variable whose contents you want to search, and *searchString* is the string for which you are searching. The *searchString* argument can be a `string` literal constant or the name of

either a `string` variable or `string` named constant. The *subscript* argument specifies the starting position for the search. In other words, it specifies the subscript of the character at which the search should begin. The **find** function searches for the *searchString* in the *string*, starting with the character whose subscript is specified in the *subscript* argument. The **find** function performs a case-sensitive search, which means that uppercase letters are not equivalent to their lowercase counterparts. When the *searchString* is contained within the *string*, the **find** function returns an integer that indicates the beginning position (subscript) of the *searchString* within the *string*. The function returns the number -1 when the *searchString* is not contained within the *string*. Figure 13-21 also shows examples of using the **find** function. The `location = phone.find("(312)", 0);` statement in Example 1 searches for the *searchString* "(312)" in the `phone` variable, beginning with the first character (whose subscript is 0) in the variable. It then assigns the result—in this case, the number 0—to the `location` variable. The number 0 is assigned because the *searchString* "(312)" begins with the character whose subscript is 0 in the `phone` variable. The `spaceLocation = name.find(" ", 1);` statement in Example 2 searches for the space character in the `name` variable, starting with the second character (whose subscript is 1). It assigns the result (5) to the `spaceLocation` variable. The number 5 is assigned because the *searchString* (`" "`) has a subscript of 5 in the `name` variable. The `location = address.find("Elm ", 2);` statement in Example 3 searches the third through the last characters in the `address` variable, looking for the string "Elm ". The statement assigns the number 4 to the `location` variable, because the string "Elm " begins with the character whose subscript is 4 in the `address` variable. The `location = address.find("elm ", 0);` statement in Example 4 searches for the string "elm " in the `address` variable, starting with the first character. The statement assigns the number -1 to the `location` variable, because the `address` variable does not contain the string "elm ". The `location = address.find("Elm ", 9);` statement in Example 5 searches for the string "Elm " in the tenth through the last characters in the `address` variable. The statement assigns the number -1 to the `location` variable, because the string "Elm " does not appear in the tenth through the last characters in the `address` variable. In other words, it doesn't appear in the characters "treet, Elmwood, NJ".

HOW TO Use the **find** Function

Syntax

string.**find(***searchString*, *subscript***)**

Example 1

```
int location = 0;
string phone = "(312) 999-9999";
location = phone.find("(312)", 0);
```

searches the `phone` variable, starting with the first character (subscript 0), to determine the location of the string "(312)"; stores the result (0) in the `location` variable

Figure 13-21 How to use the **find** function *(continues)*

(continued)

Example 2

```
int spaceLocation = 0;
string name = "Carol Cho";
spaceLocation = name.find(" ", 1);
```

searches the `name` variable, starting with the second character (subscript 1), to determine the location of the space character; stores the result (5) in the `spaceLocation` variable

Example 3

```
int location = 0;
string address = "210 Elm Street, Elmwood, NJ";
location = address.find("Elm ", 2);
```

searches the `address` variable, starting with the third character (subscript 2), to determine the location of the string "Elm "; stores the result (4) in the `location` variable

Example 4

```
int location = 0;
string address = "210 Elm Street, Elmwood, NJ";
location = address.find("elm ", 0);
```

searches the `address` variable, starting with the first character (subscript 0), to determine the location of the string "elm "; stores the result (-1) in the `location` variable

Example 5

```
int location = 0;
string address = "210 Elm Street, Elmwood, NJ";
location = address.find("Elm ", 9);
```

searches the `address` variable, starting with the tenth character (subscript 9), to determine the location of the string "Elm "; stores the result (-1) in the `location` variable



Notice the space after the letter m in the **find** function in Examples 3, 4, and 5 in Figure 13-21.

Figure 13-21 How to use the **find** function

Figure 13-22 shows the code for the rearranged name program. The **find** function appears on Line 23 and is shaded in the figure. After the **find** function determines the location of the space, the program uses the location in the two `substr` functions to separate the first name from the last name. The `substr` functions appear on Lines 24 and 25 and are shaded in the figure. Figure 13-23 shows a sample run of the program.

```
1  //Rearranged Name.cpp
2  //Displays the last name followed by a comma,
3  //a space, and the first name
4  //Created/revised by <your name> on <current date>
5
6  #include <iostream>
7  #include <string>
```

Figure 13-22 Rearranged name program

(continued)

```
 8  using namespace std;
 9
10  int main()
11  {
12      //declare variables
13      string firstLast   = "";
14      string first       = "";
15      string last        = "";
16      int spaceLocation = 0;
17
18      //get first and last name
19      cout << "Name (first and last): ";
20      getline(cin, firstLast);
21      //locate space, then pull out first and
22      //last names
23      spaceLocation = firstLast.find(" ", 0);
24      first = firstLast.substr(0, spaceLocation);
25      last = firstLast.substr(spaceLocation + 1);
26
27      //display rearranged name
28      cout << last << ", " << first << endl;
29
30      system("pause");
31      return 0;
32  }   //end of main function
```

your C++ development tool may not require this statement

Figure 13-22 Rearranged name program

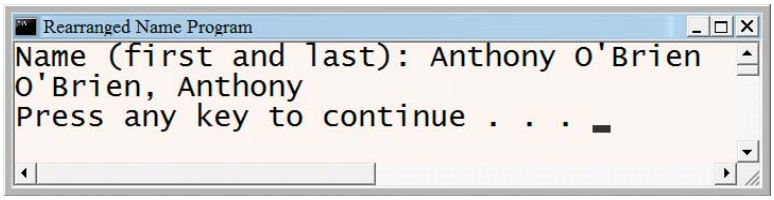


Figure 13-23 Sample run of the rearranged name program

The Annual Income Program

Figure 13-24 shows the problem description and IPO chart for the annual income program. The program should allow the user to enter a company's annual income. It then should remove any commas and spaces from the user's entry before displaying the annual income on the screen. You will learn how to remove characters from a `string` variable in the next section.

Problem specification
Create a program that allows the user to enter a company's annual income. The program should remove any commas and spaces from the annual income before displaying it on the screen.

Figure 13-24 Problem specification and IPO chart for the annual income program *(continues)*

(continued)

Input	Processing	Output
annual income	Processing items: character subscript (0) Algorithm: 1. enter the annual income 2. repeat while (the annual income is not "-1") repeat while (the character subscript is less than the length of the annual income) if (the current character in the annual income is either a comma or a space) remove the current character from the annual income else add 1 to the character subscript to look at the next character in the annual income end if end repeat display the annual income with no commas or spaces enter the annual income reset the character subscript to 0 end repeat	annual income with no commas or spaces

Figure 13-24 Problem specification and IPO chart for the annual income program

Removing Characters from a `string` Variable

At times, you may need to remove one or more characters from an item of data entered by the user, such as a dollar sign from the beginning of a sales amount or a percent sign from the end of a tax rate. In C++, you can use the `string` class's **erase function** to remove one or more characters located anywhere in a `string` variable. Figure 13-25 shows the function's syntax. In the syntax, *string* is the name of a `string` variable, and the *subscript* argument is the subscript of the first character you want to remove (erase) from the variable's contents. Recall that the first character in a string has a subscript of 0. The optional *count* argument is an integer that specifies the number of characters you want removed. If you omit the *count* argument, the `erase` function removes all characters from the *subscript* position through the end of the string. Figure 13-25 also shows examples of using the `erase` function. The `place.erase(0, 7);` statement in Example 1 tells the computer to remove the first seven characters from the string stored in the `place` variable. In this case, the computer removes the letters S, a, l, e, and m and the comma and space characters. After the statement is processed, the `place` variable contains the string "Oregon". The `place.erase(5);` statement in Example 2 tells the computer to remove all of the characters from the `place` variable, beginning with the character whose subscript is 5. In this case, the statement removes the ", Oregon" portion of the string from the variable.

After the statement is processed, the `place` variable contains the string "Salem". The `name.erase(2, 1);` statement in Example 3 removes one character, beginning with the character whose subscript is 2, from the string stored in the `name` variable. In other words, it removes the letter h. After the statement is processed, the `name` variable contains the string "Jon". The code in Example 4 contains a loop that looks at each character in the `salary` variable, one at a time. The condition in the `if` statement within the loop compares the current character to both a comma and a space. If the current character is either of those characters, the `salary.erase(subscript, 1);` statement in the `if` statement's true path removes the character from the variable. Otherwise, the `subscript += 1;` statement increments the `subscript` variable by 1, which allows the loop to look at the next character in the `salary` variable.

HOW TO Use the `erase` Function

Syntax

string.**erase(***subscript*[**,** *count*]**);**

Example 1

```
string place = "Salem, Oregon";
place.erase(0, 7);
```

removes the first seven characters from the `place` variable, changing the variable's contents to "Oregon"

Example 2

```
string place = "Salem, Oregon";
place.erase(5);
```

removes all of the characters from the `place` variable, beginning with the character whose subscript is 5, changing the variable's contents to "Salem"

Example 3

```
string name = "John";
name.erase(2, 1);
```

removes the third character from the `name` variable, changing the variable's contents to "Jon"

Example 4

```
int subscript = 0;
string salary = "";
cout << "Salary: ";
getline(cin, salary);
while (subscript < salary.length())
     if (salary.substr(subscript, 1) == ","
          || salary.substr(subscript, 1) == " ")
          salary.erase(subscript, 1);
     else
          subscript += 1;
     //end if
//end while
```

you also can use `subscript++;`

removes (erases) any commas and spaces from the `salary` variable

Figure 13-25 How to use the `erase` function

Figure 13-26 shows the annual income program. The `erase` function appears on Line 24 and is shaded in the figure. Figure 13-27 shows a sample run of the program.

```
 1  //Annual Income.cpp - displays the annual
 2  //income with any commas and spaces removed
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  #include <string>
 7  using namespace std;
 8
 9  int main()
10  {
11      string income = "";
12      int subscript = 0;
13
14      cout << "Annual income (-1 to end): ";
15      getline(cin, income);
16
17      while (income != "-1")
18      {
19          //remove any commas or spaces
20          while (subscript < income.length())
21          {
22               if (income.substr(subscript, 1) == ","
23                    || income.substr(subscript, 1) == " ")
24                         income.erase(subscript, 1);
25               else
26                    subscript += 1;
27               //end if
28          }    //end while
29
30          //display annual income
31          cout << "Annual income with no commas "
32               << "or spaces: $" << income << endl << endl;
33
34          cout << "Annual income (-1 to end): ";
35          getline(cin, income);
36          subscript = 0;
37      }   //end while
38
39      system("pause");
40      return 0;
41   }  //end of main function
```

you also can use `subscript++;`

your C++ development tool may not require this statement

Figure 13-26 Annual income program

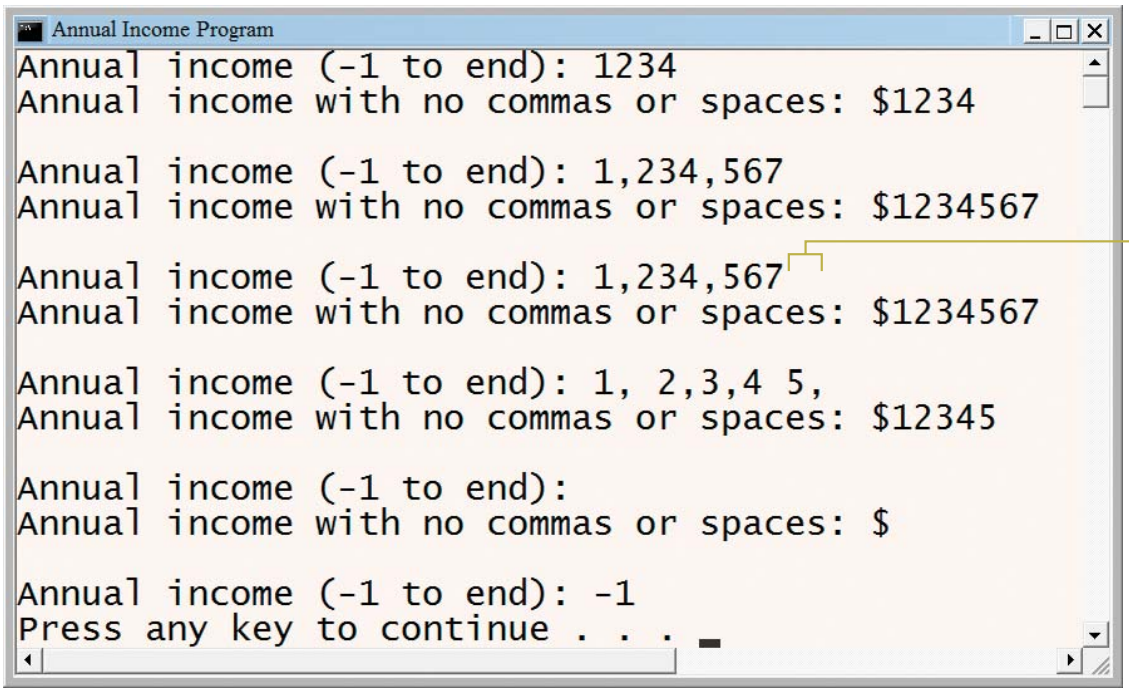

Annual Income Program
Annual income (-1 to end): 1234
Annual income with no commas or spaces: $1234
Annual income (-1 to end): 1,234,567
Annual income with no commas or spaces: $1234567
Annual income (-1 to end): 1,234,567
Annual income with no commas or spaces: $1234567
Annual income (-1 to end): 1, 2,3,4 5,
Annual income with no commas or spaces: $12345
Annual income (-1 to end):
Annual income with no commas or spaces: $
Annual income (-1 to end): -1
Press any key to continue . . .
2 spaces

Figure 13-27 Sample run of the annual income program

Replacing Characters in a `string` Variable

Rather than using the `erase` function to code the annual income program from the previous section, you also can use the `string` class's `replace` function. The **replace function** replaces a sequence of characters in a `string` variable with another sequence of characters. For example, you can use the `replace` function to replace area code "800" with area code "877" in a phone number. Or, you can use it to replace the dashes in a Social Security number with the empty string. You also can use it to `replace` a character, such as a comma or a space, with the empty string. Figure 13-28 shows the `replace` function's syntax and includes examples of using the function. In the syntax, *string* is the name of a `string` variable, and the *subscript* argument specifies where to begin replacing characters in the *string*. The *count* argument indicates the number of characters to replace, and the *replacementString* argument contains the replacement characters. The `phone.replace(2, 3, "877");` statement in Example 1 tells the computer to replace the "800" in the `phone` variable with "877". After the statement is processed, the `phone` variable contains 1-877-111-0000. The `item.replace(3, 1, "D");` statement in Example 2 replaces the letter X, whose subscript is 3 in the `item` variable, with the letter "D". In other words, the statement changes the value stored in the `item` variable from "ABCX34" to "ABCD34". The `name.replace(7, 6, "Farley");` statement in Example 3 replaces the string "Wilson" in the `name` variable with the string "Farley"; doing this changes the variable's contents to "Karena Farley". Example 4 replaces with the empty string any commas and spaces contained in the `salary` variable. In other words, it removes the commas and spaces from the variable.

HOW TO Use the `replace` Function

Syntax

string.**replace(***subscript*, *count*, *replacementString***);**

Example 1

```
string phone = "1-800-111-0000";
phone.replace(2, 3, "877");
```

replaces three characters in the `phone` variable, beginning with the character whose subscript is 2, with "877"; changes the contents of the `phone` variable to "1-877-111-0000"

Example 2

```
string item = "ABCX34";
item.replace(3, 1, "D");
```

replaces one character in the `item` variable, beginning with the character whose subscript is 3, with "D"; changes the contents of the `item` variable to "ABCD34"

Example 3

```
string name = "Karena Wilson";
name.replace(7, 6, "Farley");
```

replaces six characters in the `name` variable, beginning with the character whose subscript is 7, with "Farley"; changes the contents of the `name` variable to "Karena Farley"

Example 4

```
int subscript = 0;
string salary = "";
cout << "Salary: ";
getline(cin, salary);
while (subscript < salary.length())
     if (salary.substr(subscript, 1) == ","
          || salary.substr(subscript, 1) == " ")
          salary.replace(subscript, 1, "");
     else
          subscript += 1;
     //end if
//end while
```

you also can use `subscript++;`

replaces with the empty string any commas and spaces in the `salary` variable

Figure 13-28 How to use the `replace` function

Figure 13-29 shows how you can use the `replace` function, instead of the `erase` function, in the annual income program. The modification made to the original code (shown earlier in Figure 13-26) is shaded in Figure 13-29.

```
19          //remove any commas or spaces
20          while (subscript < income.length())
21          {
22               if (income.substr(subscript, 1) == ","
23                    || income.substr(subscript, 1) == " ")
24                         income.replace(subscript, 1, "");
25               else
26                    subscript += 1;
27               //end if
28          }    //end while
```

you also can use `subscript++;`

Figure 13-29 Partial annual income program showing the `replace` function

Mini-Quiz 13-3



The answers to Mini-Quiz questions are located in Appendix A.

1. Which of the following searches for a comma in a `string` variable named `cityState` and then assigns the result to an `int` variable named `location`?

   a. `location = cityState.find(",", 0);`

   b. `location = cityState.find(0, ",");`

   c. `location = cityState.search(",", 0);`

   d. `location = cityState.searchFor(",");`

2. If the `cityState` variable contains the string "Bowling Green, KY", what will the statement from Question 1 assign to the `location` variable?

3. If the `cityState` variable contains the string "Bowling Green, KY", which of the following changes the variable's contents to "Bowling Green"?

   a. `cityState.erase(13);`

   b. `cityState.erase(13, 4);`

   c. `cityState.replace(13, 4, "");`

   d. all of the above

The Social Security Number Program

Figure 13-30 shows the problem description and IPO chart for the Social Security number program. The program should allow the user to enter a Social Security number without the hyphens. If the user's entry contains nine characters, the program should insert hyphens in the appropriate places in the Social Security number and then display the result on the screen. If the user did not enter nine characters, the program should display an appropriate message. You will learn how to insert characters within a `string` variable in the next section.

Problem specification
Create a program that allows the user to enter a Social Security number without the two hyphens. The user's entry should contain nine characters. If the user did not enter the required number of characters, the program should display the message "The number must contain 9 characters". Otherwise, the program should insert the two missing hyphens and then display the result on the screen.

Input	Processing	Output
SSN (without hyphens)	Processing items: none	SSN (with hyphens)

```
Algorithm:
1. enter the SSN
2. if (the SSN contains 9 characters)
       insert a hyphen as the fourth
       character in the SSN

       insert a hyphen as the seventh
       character in the SSN

       display the SSN
   else
       display the message "The number
       must contain 9 characters"
   end if
```

Figure 13-30 Problem specification and IPO chart for the Social Security number program

Inserting Characters Within a `string` Variable

In addition to removing and replacing characters in a `string` variable, you also can insert characters within a `string` variable. For example, you can insert an employee's middle initial between his or her first and last names. Or, you can insert parentheses around the area code in a phone number. You also can insert hyphens in a Social Security number. The `string` class provides the **insert function** for inserting characters within a `string` variable. The function's syntax is shown in Figure 13-31. In the syntax, *string* is the name of a `string` variable, and *subscript* specifies where in the *string* you want the *insertString* inserted. To insert the *insertString* at the beginning of the *string*, you use the number 0 as the *subscript*. To insert the *insertString* starting with the second character in the *string*, you use the number 1 as the *subscript*, and so on. Also shown in Figure 13-31 are examples of using the `insert` function. The `name.insert(7, "G. ");` statement in Example 1 tells the computer to insert the *insertString*—in this case, "G. " (the letter G, a period, and a space)—in the `name` variable. The letter G is inserted in subscript 7, which makes it the eighth character in the `name` variable. The period and space are inserted in subscripts 8 and 9, making them the ninth and tenth characters in the variable. After the statement is processed, the `name` variable contains the string "Harold G. Cruthers". In Example 2, the `phone.insert(0, "(");` statement changes the contents of the `phone` variable from "312 050-1111" to "(312 050-1111". The `phone.insert(4, ")");` statement then changes the variable's contents to "(312) 050-1111". The `ssn.insert(3, "-");` statement in Example 3 changes the contents

of the `ssn` variable from "111220000" to 111-220000". The `ssn.insert(6, "-");` statement then changes the variable's contents to "111-22-0000".

HOW TO Use the `insert` Function

Syntax

string.**insert(***subscript*, *insertString***);**

Example 1

```
string name = "Harold Cruthers";
name.insert(7, "G. ");
```

inserts the letter G, followed by a period and a space, between the first and last `names` stored in the `name` variable; changes the contents of the `name` variable to "Harold G. Cruthers"

Example 2

```
string phone = "312 050-1111";
phone.insert(0, "(");
phone.insert(4, ")");
```

inserts the opening and closing parentheses at the beginning and end, respectively, of the area code; changes the contents of the `phone` variable to "(312) 050-1111"

Example 3

```
string ssn = "111220000";
ssn.insert(3, "-");
ssn.insert(6, "-");
```

inserts two hyphens in the Social Security number, one after the third number and the other after the fifth number; changes the contents of the `ssn` variable to "111-22-0000"

Figure 13-31 How to use the `insert` function

Figure 13-32 shows the Social Security number program. The `insert` function appears on Lines 17 and 18; both lines are shaded in the figure. Figure 13-33 shows a sample run of the program.

```
 1  //Social Security Number.cpp
 2  //Displays the Social Security number with hyphens
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  #include <string>
 7  using namespace std;
 8
 9  int main()
10  {
11        string ssn = "";
12        cout << "Social Security number without hyphens: ";
```

Figure 13-32 Social Security number program *(continues)*

(continued)

```
13      getline(cin, ssn);
14
15      if (ssn.length() == 9)
16      {
17          ssn.insert(3, "-");
18          ssn.insert(6, "-");
19          cout << "Social Security number: " << ssn << endl;
20      }
21      else
22          cout << "The number must contain "
23          << "9 characters" << endl;
24      //end if
25
26      system("pause");
27      return 0;
28  }   //end of main function
```

your C++ development tool may not require this statement

Figure 13-32 Social Security number program

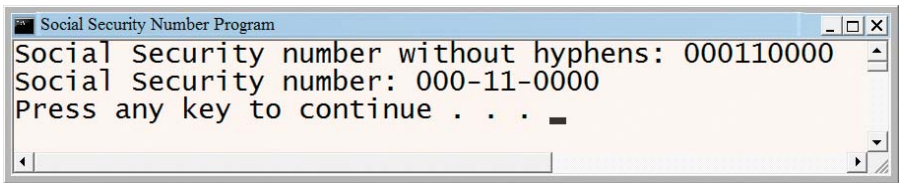

Social Security Number Program
Social Security number without hyphens: 000110000
Social Security number: 000-11-0000
Press any key to continue . . .

Figure 13-33 Sample run of the Social Security number program

The Company Name Program

Figure 13-34 shows the problem description and IPO chart for the company name program. The program should allow the user to enter the name of a company. It then should display the name with a row of hyphens below it.

Problem specification
Create a program that allows the user to enter the name of a company. The program should display the name underlined with a row of hyphens. The number of hyphens should be the same as the number of characters in the company name.

Input	Processing	Output
company name	Processing items: none Algorithm: 1. enter the company name 2. display the company name 3. display a row of hyphens below the company name	company name row of hyphens

Figure 13-34 Problem specification and IPO chart for the company name program

Duplicating a Character Within a `string` Variable

You can use the `string` class's **assign function** to duplicate one character a specified number of times and then assign the resulting string to a `string` variable. Figure 13-35 shows the function's syntax and includes examples of using the function. In the syntax, *string* is the name of a `string` variable that will store the duplicated characters. The *count* argument is an integer that indicates the number of times you want to duplicate the character specified in the function's *character* argument. The *character* argument can be either a character literal constant enclosed in single quotation marks or the name of a `char` memory location. The `asterisks.assign(10, '*');` statement in Example 1 duplicates the asterisk character 10 times and then assigns the resulting string to the `asterisks` variable. The `underline.assign(companyName.length(), '-');` statement in Example 2 duplicates the hyphen character zero or more times, depending on the number of characters in the `companyName` variable. It then assigns the resulting string to the `underline` variable.

HOW TO Use the `assign` Function

Syntax

string.**assign(***count***,** *character***);**

Example 1

```
string asterisks = "";
asterisks.assign(10, '*');
```

assigns 10 asterisks to the `asterisks` variable

Example 2

```
string companyName = "";
string underline = "";
cout << "Company name: ";
getline(cin, companyName);
underline.assign(companyName.length(), '-');
```

assigns zero or more hyphens to the `underline` variable; the number of hyphens depends on the number of characters in the `companyName` variable

Figure 13-35 How to use the `assign` function

Figure 13-36 shows the company name program. The `assign` function appears on Line 18 and is shaded in the figure. Figure 13-37 shows a sample run of the program.

```
1  //Company Name.cpp - displays the company name
2  //underlined with a row of hyphens
3  //Created/revised by <your name> on <current date>
4
5  #include <iostream>
6  #include <string>
```

Figure 13-36 Company name program *(continues)*

(continued)

```
 7  using namespace std;
 8
 9  int main()
10  {
11      string companyName = "";
12      string underline   = "";
13
14      //get the company name
15      cout << "Company name: ";
16      getline(cin, companyName);
17      //assign the appropriate number of hyphens
18      underline.assign(companyName.length(), '-');
19      //display the company name and row of hyphens
20      cout << endl << companyName << endl;
21      cout << underline << endl;
22
23      system("pause");
24      return 0;
25  }   //end of main function
```

your C++ development tool may not require this statement

Figure 13-36 Company name program

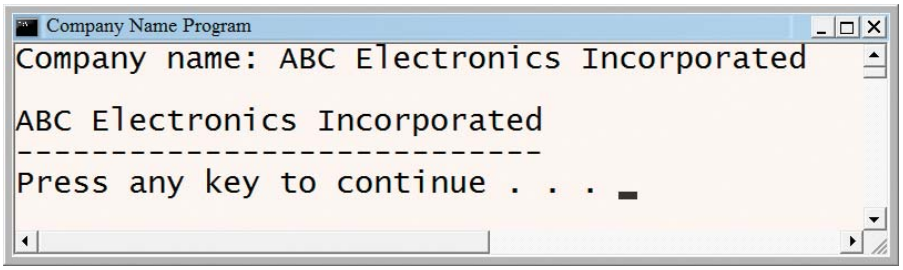

Company Name Program
Company name: ABC Electronics Incorporated
ABC Electronics Incorporated
Press any key to continue . . .

Figure 13-37 Sample run of the company name program

Concatenating Strings

The company name program, which you viewed in the previous section, used the `assign` function to assign zero or more hyphens to a `string` variable named `underline`. You can accomplish the same result using string concatenation (rather than the `assign` function). **String concatenation** refers to the process of connecting (or linking) strings together. You concatenate strings using the **concatenation operator,** which is the + sign in C++. Figure 13-38 shows examples of using the concatenation operator in a C++ statement. The `full = first + " " + last;` statement in Example 1 concatenates the contents of the `first` variable, a space, and the contents of the `last` variable. It then assigns the concatenated string ("Sydney Holmes") to the `full` variable. The `sentence = sentence + "?";` statement in Example 2 concatenates the contents of the `sentence` variable and a question mark (?) and then assigns the concatenated string ("How are you?") to the `sentence` variable. The `underline = underline + "-";` statement in Example 3 appears within a loop that will repeat the statement for each character in the `companyName` variable. The statement concatenates a hyphen (-) and the

current contents of the `underline` variable and then assigns the result to the `underline` variable.

HOW TO Use the Concatenation Operator

Example 1

```
string first = "Sydney";
string last = "Holmes";
string full = "";
full = first + " " + last;
```

concatenates the contents of the `first` variable, a space, and the contents of the `last` variable and then assigns the result ("Sydney Holmes") to the `full` variable

Example 2

```
string sentence = "How are you";
sentence = sentence + "?";
```

concatenates the contents of the `sentence` variable and a question mark and then assigns the result ("How are you?") to the `sentence` variable

Example 3

```
string companyName = "";
string underline = "";
cout << "Company name: ";
getline(cin, companyName);
for (int x = 1; x <= companyName.length(); x += 1)
     underline = underline + "-";
//end for
```

concatenates zero or more hyphens within the `underline` variable; the number of hyphens depends on the number of characters in the `companyName` variable

Figure 13-38 How to use the concatenation operator

Figure 13-39 shows how you can use string concatenation, instead of the `assign` function, in the company name program. The modifications made to the original code (shown earlier in Figure 13-36) are shaded in Figure 13-39. Although you could use the loop shown in Figure 13-39 to assign the appropriate number of hyphens to the `underline` variable, it is much easier to use the `underline.assign(companyName.length(), '-');` statement.

```
17      //assign the appropriate number of hyphens
18      for (int x = 1; x <= companyName.length(); x += 1)
19          underline = underline + "-";
20      //end for
21      //display the company name and row of hyphens
22      cout << endl << companyName << endl;
23      cout << underline << endl;
```



You also can use `x++` in Line 18, and `underline += "-";` in Line 19.

Figure 13-39 Partial company name program showing string concatenation



The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 13-4

1. Which of the following changes the contents of the `cityState` variable from "Las Vegas Nevada" to "Las Vegas, Nevada"?

   a. `cityState.insert(10, ",");`

   b. `cityState.replace("s N", "s, N");`

   c. `cityState.assign(9, ",");`

   d. none of the above

2. The `temp` and `sentence` variables are `string` variables. Which of the following assigns four exclamation points to the `temp` variable and then concatenates the variable and the `sentence` variable?

   a. `sentence = sentence + temp.assign(4, '!');`

   b. `sentence = sentence + temp.assign(4, "!");`

   c. `sentence = sentence + temp.assign('!', 4);`

   d. none of the above

3. Which of the following concatenates the opening parentheses, the contents of the `areaCode` variable, and the closing parentheses and then assigns the result to the `areaCode` variable?

   a. `areaCode = "(" + "areaCode" + ")";`

   b. `areaCode = "(" + areaCode + ")";`

   c. `areaCode = '(' & areaCode & ')';`

   d. none of the above



The answers to the labs are located in Appendix A.



LAB 13-1 Stop and Analyze

Study the program shown in Figure 13-40 and then answer the questions.

```
1  //Lab13-1.cpp
2  //Removes parentheses and hyphens from a phone number
3  //Created/revised by <your name> on <current date>
4
5  #include <iostream>
6  #include <string>
7  using namespace std;
8
```

Figure 13-40 Code for Lab 13-1 *(continues)*

(continued)

```
 9  int main()
10  {
11      string phone       = "";
12      string currentChar = "";
13      int numChars       = 0;
14      int subscript      = 0;
15
16      //get phone number
17      cout << "Enter a phone number: ";
18      getline(cin, phone);
19
20      //determine number of characters
21      numChars = phone.length();
22
23      //remove any parentheses or hyphens
24      while (subscript < numChars)
25      {
26          currentChar = phone.substr(subscript, 1);
27          if (currentChar == "("
28              || currentChar == ")"
29              || currentChar == "-")
30          {
31              phone.erase(subscript, 1);
32              numChars = numChars - 1;
33          }
34          else
35              subscript += 1;
36          //end if
37      }   //end while
38
39      //display phone number
40      cout << "Phone number: " << phone << endl;
41
42      system("pause");
43      return 0;
44  }   //end of main function
```

your C++ development tool may not require this statement



You also can use either `numChars -= 1;` or `numChars--;` in Line 32, and `subscript++;` in Line 35.

Figure 13-40 Code for Lab 13-1

QUESTIONS

1. What is the purpose of the loop in Lines 24 through 37?
2. What is the purpose of the statement in Line 26?
3. What is the purpose of the selection structure in Lines 27 through 36?
4. Why is the statement in Line 32 necessary?
5. Why is the statement in Line 35 processed only when the current character is not the opening parentheses, closing parentheses, or hyphen? In other words, why isn't it necessary to update the `subscript` variable when the character is the opening parentheses, closing parentheses, or hyphen?

6. Follow the instructions for starting C++ and opening the Lab13-1.cpp file. The file is contained in either the Cpp6\Chap13\Lab13-1 Project folder or the Cpp6\Chap13 folder. Run the program. When you are prompted to enter a phone number, type (312)050-1234 and press Enter. The program removes the parentheses and hyphen from the phone number and then displays 3120501234 on the screen.
7. Modify the program to use the `replace` function rather than the `erase` function. Save and then run the program.



LAB 13-2 Plan and Create

In this lab, you will plan and create an algorithm for Mr. Coleman. The problem specification, IPO chart information, and C++ instructions are shown in Figure 13-41.

Problem specification

Mr. Coleman teaches second grade at Hinsbrook School. On days when the weather is bad and the students cannot go outside to play, he spends recess time playing a simplified version of the Hangman game with his class. The game requires two people to play. Currently, Mr. Coleman thinks of a word that has five letters. He then draws five dashes on the chalkboard—one for each letter in the word. One student then is chosen to guess the word, letter by letter. When the student guesses a correct letter, Mr. Coleman replaces the appropriate dash or dashes with the letter. For example, if the original word is *moose* and the student guesses the letter *o*, Mr. Coleman changes the five dashes on the chalkboard to -*oo*--. The game is over when the student either guesses all of the letters in the word or makes 10 incorrect guesses, whichever occurs first. Mr. Coleman wants a program that allows two students to play the game on the computer.

IPO chart information	C++ instructions
Input	
original word (from player 1)	`string origWord = "";`
letter (from player 2)	`string letter = "";`
Processing	
variable that keeps track of whether a dash was replaced ('N')	`char dashReplaced = 'N';`
variable that keeps track of whether the game is over ('N')	`char gameOver = 'N';`
number of incorrect guesses	`int numIncorrect = 0;`
Output	
display word (5 dashes when the program begins)	`string displayWord = "-----";`

Figure 13-41 Problem specification, IPO chart information, and C++ instructions for Lab 13-2 *(continues)*

(continued)

Algorithm

```
1. repeat                                   do //begin loop
                                            {
      get original word                        cout << "Enter a 5-letter word
                                               in uppercase: ";
                                               getline(cin, origWord);
   while (the original word                 }  while (origWord.length() != 5);
   does not contain exactly
   five characters)
2. clear the screen                         system("cls");
3. display the five dashes contained        cout << "Guess this word: " <<
   in the display word                      displayWord << endl;
4. repeat while (the game is not over)      while (gameOver == 'N')
                                            {
      get an uppercase letter                  cout << "Enter an uppercase
                                               letter: ";
                                               cin >> letter;

      repeat for (each letter in the            for (int x = 0; x < 5; x += 1)
      original word)                            {
         if (the current character in the          if (origWord.substr(x, 1)
         original word matches the letter)         == letter)
                                                   {
            replace the dash in the                   displayWord.replace(x,
            display word with the letter              1, letter);

            assign 'Y' to the variable                dashReplaced = 'Y';
            that keeps track of whether
            a dash was replaced
         end if                                    }  //end if
      end repeat                                }  //end for

      if (a dash was replaced)                  if (dashReplaced == 'Y')
                                                {
         if (the display word does not             if (displayWord.find("-",
         contain any dashes)                       0) == -1)
                                                   {
            assign 'Y' to the variable                gameOver = 'Y';
            that keeps track of whether
            the game is over

            display the original word                 cout << endl << "Yes, the
                                                      word is " << origWord
                                                      << endl;
            display "Great guessing"                  cout << "Great guessing!"
            message                                   << endl;
                                                   }
         else                                      else
                                                   {
            display the status of the                 cout << endl << "Guess this
            display word                              word: " << displayWord
                                                      << endl;
```

Figure 13-41 Problem specification, IPO chart information, and C++ instructions for Lab 13-2 *(continues)*

(continued)

```
        reset to 'N' the variable                    dashReplaced = 'N';
        that keeps track of whether
        a dash was replaced
     end if                                       } //end if
                                                }
  else                                          else
                                                {
    add 1 to the number of                         numIncorrect += 1;
    incorrect guesses
    if (the number of incorrect                    if (numIncorrect == 10)
    guesses is 10)                                 {
      assign 'Y' to the variable                      gameOver = 'Y';
      that keeps track of whether
      the game is over

      display "Sorry, the word is"                 cout << endl <<
      and the original word                        "Sorry, the word
                                                   is " << origWord
                                                   << endl;
     end if                                        } //end if
  end if                                        } //end if
end repeat                                    } //end while
```

Figure 13-41 Problem specification, IPO chart information, and C++ instructions for Lab 13-2

Figure 13-42 shows the code for the entire Hangman game program, and Figures 13-43 and 13-44 show sample runs of the program.

```
 1  //Lab13-2.cpp - simulates the Hangman game
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <string>
 6  using namespace std;
 7
 8  int main()
 9  {
10     //declare variables
11     string origWord    = "";
12     string letter      = "";
13     char dashReplaced  = 'N';
14     char gameOver      = 'N';
15     int numIncorrect   = 0;
16     string displayWord = "-----";
17
18     //get original word
19     do //begin loop
20     {
21         cout << "Enter a 5-letter word in uppercase: ";
22         getline(cin, origWord);
23     }   while (origWord.length() != 5);
24
```

Figure 13-42 Hangman game program *(continues)*

(continued)

```
25     //clear the screen
26     system("cls");
27
28     //start guessing
29     cout << "Guess this word: " <<
30     displayWord << endl;
31     while (gameOver == 'N')
32     {
33        cout << "Enter an uppercase letter: ";
34        cin >> letter;
35
36        //search for the letter in the original word
37        for (int x = 0; x < 5; x += 1)
38        {
39            //if the current character matches
40            //the letter, replace the corresponding
41            //dash in the displayWord variable and then
42            //set the dashReplaced variable to 'Y'
43            if (origWord.substr(x, 1) == letter)
44            {
45                displayWord.replace(x, 1, letter);
46                dashReplaced = 'Y';
47            }   //end if
48         }  //end for
49
50        //if a dash was replaced, check whether the
51        //displayWord variable contains any dashes
52        if (dashReplaced == 'Y')
53        {
54             //if the displayWord variable does not
55             //contain any dashes, the game is over
56             if (displayWord.find("-", 0) == -1)
57             {
58                gameOver = 'Y';
59                cout << endl << "Yes, the word is "
60                    << origWord << endl;
61                cout << "Great guessing!" << endl;
62             }
63             else //otherwise, continue guessing
64             {
65                 cout << endl << "Guess this word: "
66                    << displayWord << endl;
67                dashReplaced = 'N';
68             }  //end if
69        }
70        else //processed when dashReplaced contains 'N'
71        {
72            //add 1 to the number of incorrect guesses
73            numIncorrect += 1;
74            //if the number of incorrect guesses is 10,
75            //the game is over
76            if (numIncorrect == 10)
77            {
78                gameOver = 'Y';
79                cout << endl << "Sorry, the word is "
80                    << origWord << endl;
81            }  //end if
```

Figure 13-42 Hangman game program *(continues)*

(continued)

```
82          }  //end if
83      }  //end while
84
85      system("pause");
86      return 0;
87  }  //end of main function
```

your C++ development tool may not require this statement

Figure 13-42 Hangman game program

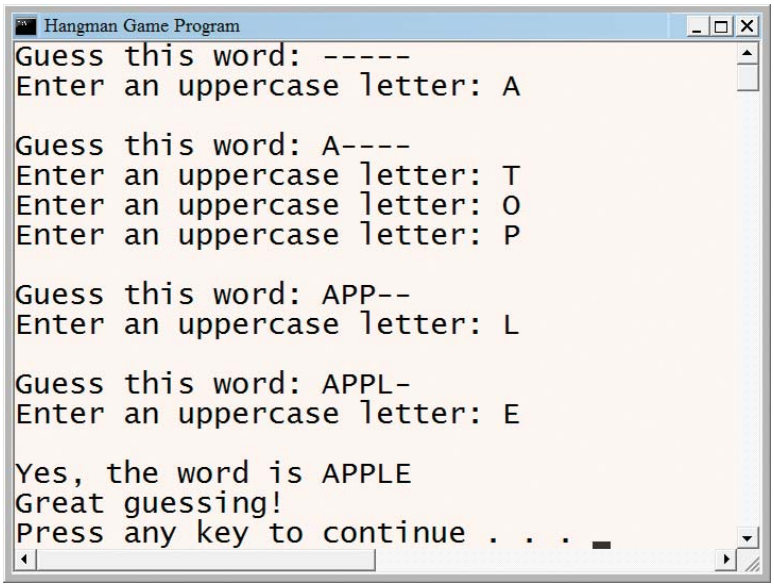

Hangman Game Program
Guess this word: -----
Enter an uppercase letter: A
Guess this word: A----
Enter an uppercase letter: T
Enter an uppercase letter: O
Enter an uppercase letter: P
Guess this word: APP--
Enter an uppercase letter: L
Guess this word: APPL-
Enter an uppercase letter: E
Yes, the word is APPLE
Great guessing!
Press any key to continue . . .

Figure 13-43 Sample run of the Hangman game program

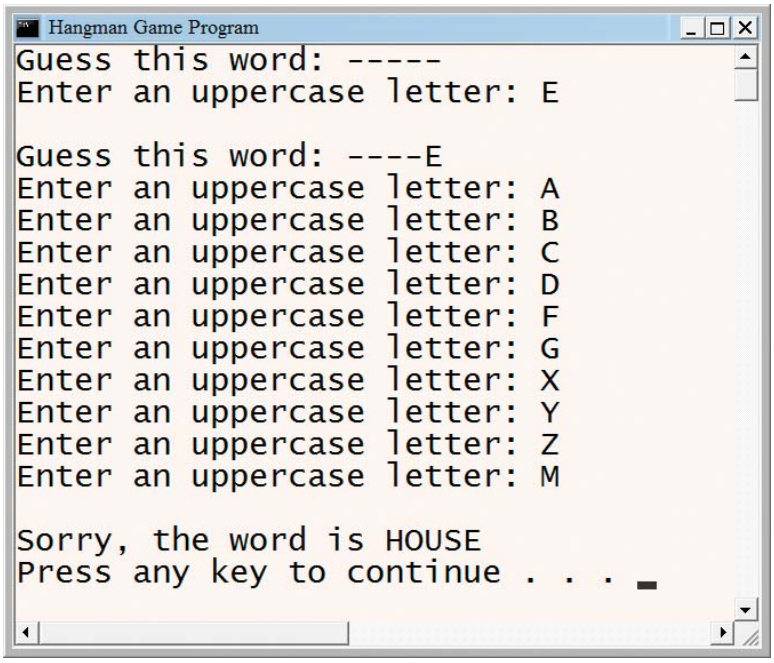

Hangman Game Program
Guess this word: -----
Enter an uppercase letter: E
Guess this word: ----E
Enter an uppercase letter: A
Enter an uppercase letter: B
Enter an uppercase letter: C
Enter an uppercase letter: D
Enter an uppercase letter: F
Enter an uppercase letter: G
Enter an uppercase letter: X
Enter an uppercase letter: Y
Enter an uppercase letter: Z
Enter an uppercase letter: M
Sorry, the word is HOUSE
Press any key to continue . . .

Figure 13-44 Another sample run of the Hangman game program

DIRECTIONS

Follow the instructions for starting your C++ development tool. Depending on the development tool you are using, you may need to create a new project; if so, name the project Lab13-2 Project and save it in the Cpp6\Chap13 folder. Enter the instructions shown in Figure 13-42 in a source file named Lab13-2.cpp. (Do not enter the line numbers.) Save the file in either the project folder or the Cpp6\Chap13 folder. Now follow the appropriate instructions for running the Lab13-2.cpp file. Test the program using an original word that does not contain exactly five characters. Also test the program using the data shown in Figures 13-43 and 13-44. If necessary, correct any bugs (errors) in the program.



LAB 13-3 Modify

If necessary, create a new project named Lab13-3 Project. Enter (or copy) the Lab13-2.cpp instructions into a new source file named Lab13-3.cpp. Change Lab13-2.cpp in the first comment to Lab13-3.cpp. Currently, the Hangman game program allows player 1 to enter only a five-character word. Modify the program so that player 1 can enter a word of any length. Save and then run the program. Test the program appropriately.



LAB 13-4 Desk-Check

Desk-check the code shown in Figure 13-45. What will the code display on the screen?

```
 1  //Lab13-4.cpp - displays a message
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <string>
 6  using namespace std;
 7
 8  int main()
 9  {
10      string message     = "praogxwrazingmun";
11      string subMessage1 = "";
12      string subMessage2 = "";
13
14      message.erase(5, 2);
15      message.insert(11, "is");
16      message.replace(13, 1, "f");
17
18      subMessage1 = message.substr(0, 11);
19      subMessage1.replace(7, 1, "mm");
20      subMessage1.erase(2, 2);
21      subMessage1.replace(0, 1, "P");
22      subMessage1.insert(2, "o");
```

Figure 13-45 Code for Lab 13-4 *(continues)*

(continued)

```
23
24      subMessage2 = subMessage2.assign(5, '!');
25      subMessage2 = message.substr(11) + subMessage2;
26      subMessage2.insert(0, " ");
27      subMessage2.insert(3, " ");
28
29      message = subMessage1 + subMessage2;
30
31      //display message
32      cout << "Message: " << message << endl;
33
34      system("pause");
35      return 0;
36  }   //end of main function
```

your C++ development tool may not require this statement

Figure 13-45 Code for Lab 13-4



LAB 13-5 Debug

Follow the instructions for starting C++ and opening the Lab13-5.cpp file. The file is contained in either the Cpp6\Chap13\Lab13-5 Project folder or the Cpp6\Chap13 folder. Type Joe and press Enter. Rather than displaying the letters J, o, and e on three separate lines, the program displays Joe, oe, and e. Stop and then debug the program.

Summary

- The `string` data type was added to the C++ language using the `string` class.
- Memory locations (variables and named constants) whose data type is `string` are initialized using string literal constants, which are zero or more `characters` enclosed in double quotation marks. Most string variables are initialized to the empty string.
- You can use the extraction operator to get a string from the user at the keyboard, but only if the string does not contain a white-space character (blank, tab, or newline).
- The `getline` function gets a string of characters entered at the keyboard and stores them in a `string` variable. The string can contain any characters, including white-space characters (blanks, tabs, and newlines). The `getline` function stops reading and storing characters when it encounters the delimiter character in the input. The function's default delimiter character is the newline character. The function reads and then consumes (discards) the delimiter character.
- The computer stores the characters entered at the keyboard in the `cin` object. Both the extraction operator and the `getline` function remove

characters from the object. However, unlike the extraction operator, which leaves the newline character in the `cin` object, the `getline` function consumes the newline character.

- The `ignore` function reads and then consumes characters entered at the keyboard. The function stops reading and consuming characters when it consumes either a specified number of characters or the delimiter character, whichever occurs first. The default number of characters to consume is 1.
- Figure 13-46 shows the syntax and purpose of each function covered in the chapter. It also includes the string concatenation operator. The `assign`, `erase`, `insert`, and `replace` functions are self-contained statements that change the value of the *string* variable.

Function/Operator	Syntax	Purpose
+		concatenate strings
`assign` function	*string*.**assign(***count*, *character***);**	duplicate a character within a `string` variable
`erase` function	*string*.**erase(***subscript*[, *count*]**);**	remove one or more characters located anywhere in a `string` variable
`find` function	*string*.**find(***searchString*, *subscript***)**	search a `string` variable to determine whether it contains a specific sequence of characters
`getline` function	**getline(cin,** *stringVariableName*[, *delimeterCharacter*]**);**	get string input from the keyboard
`ignore` function	**cin.ignore(**[*numberOfCharacters*] [, *delimiterCharacter*]**);**	read and consume characters entered at the keyboard
`insert` function	*string*.**insert(***subscript*, *insertString***);**	insert characters within a `string` variable
`length` function	*string*.**length()**	determine the number of characters contained in a `string` variable
`replace` function	*string*.**replace(***subscript*, *count*, *replacementString***);**	replace a sequence of characters in a `string` variable with another sequence of characters
`substr` function	*string*.**substr(***subscript*[, *count*]**)**	access any number of characters contained in a `string` variable

Figure 13-46 Summary of `string` functions and the concatenation operator

Key Terms

`assign` function—used to duplicate one character a specified number of times within a string

Concatenation operator—used to concatenate (connect) strings; the + sign in C++

Consuming the character—another term for discarding the character

`erase` function—used to remove (erase) characters from a string

`find` function—returns an integer that indicates the beginning position of a string within a `string` variable

`getline` function—reads characters entered at the keyboard until it encounters the delimiter character, which it consumes

`ignore` function—tells the computer to first read and then consume (discard) one or more characters

`insert` function—used to insert characters within a string

`length` function—returns the number of characters contained in a `string` variable

`replace` function—used to replace characters within a string

String concatenation—the process of connecting (or linking) strings together; accomplished with the concatenation operator

`substr` function—returns the characters you want to access from a `string` variable

Review Questions

1. Which of the following statements displays the number of characters contained in a `string` variable named `address`?

   a. `cout << address.length() << endl;`

   b. `cout << numChars(address) << endl;`

   c. `cout << length(address) << endl;`

   d. `cout << size.address << endl;`

2. Which of the following statements should a program use to get the name of any city and store it in a `string` variable named `cityName`?

   a. `cin >> cityName;`

   b. `cin(cityName);`

   c. `getline(cityName, cin);`

   d. `getline(cin, cityName);`

3. If the `amount` variable contains the string "$56.55", which of the following statements will remove the dollar sign from the variable's contents?

   a. `amount.erase("$");`

   b. `amount.erase(0, 1);`

   c. `amount = amount.substr(1);`

   d. both b and c

4. If the `state` variable contains the string "MI " (the letters M and I followed by three spaces), which of the following statements will remove the three spaces from the variable's contents?

   a. `state.erase("  ");`

   b. `state.erase(3, "");`

   c. `state.remove(2, 3);`

   d. none of the above

5. The subscript of the first character contained in a `string` variable is _____.

   a. 0 (zero)

   b. 1 (one)

6. Which of the following `if` clauses determines whether the string stored in the `part` variable begins with the letter A?

   a. `if (part.begins("A"))`

   b. `if (part.beginswith("A"))`

   c. `if (part.substr(0, 1) == "A")`

   d. `if (part.substr(1) == "A")`

7. Which of the following `if` clauses determines whether the string stored in the `part` variable ends with the letter B?

   a. `if (part.ends("B"))`

   b. `if (part.endswith("B")`

   c. `if (part.substr(part.length() - 1, 1) == "B")`

   d. none of the above

8. Which of the following statements assigns the first three characters in the `part` variable to the `code` variable?

   a. `code = part.assign(0, 3);`

   b. `code = part.substr(0, 3);`

   c. `code = part.substr(1, 3);`

   d. `code = part.substring(0, 3);`

9. If the `word` variable contains the string "Bells", which of the following statements will change the contents of the variable to "Bell"?

   a. `word.erase(word.length() - 1, 1);`

   b. `word.replace(word.length() - 1, 1, "");`

   c. `word = word.substr(0, word.length() - 1);`

   d. all of the above

10. Which of the following statements changes the contents of the `word` variable from "men" to "mean"?

    a. `word.addTo(2, "a");`

    b. `word.insert(2, "a");`

    c. `word.insert(3, "a");`

    d. none of the above

11. If the `msg` variable contains the string "Happy holidays", what will the `cout << msg.find("day", 0);` statement display on the screen?

    a. -1

    b. 0

    c. 10

    d. 11

12. If the `msg` variable contains the string "Happy holidays", what will the `location = msg.find("Day", 0);` statement assign to the `location` variable?

    a. -1

    b. 0

    c. 10

    d. 11

13. Which of the following assigns the location of the comma in the `amount` variable to an `int` variable named `loc`?

    a. `loc = amount.contains(",");`

    b. `loc = amount.substr(",");`

    c. `loc = amount.find(",", 0);`

    d. none of the above

14. Which of the following statements searches for the string "CA" in a `string` variable named `state` and then assigns the result to an `int` variable named `result`? The search should begin with the character located in subscript 5 in the `state` variable. The `state` variable's contents are uppercase.

a. `result = find(state, 5, "CA");`

b. `result = state.find(5, "CA");`

c. `result = state.find("CA", 5);`

d. `result = state.find("CA", 5, 2);`

15. If the `state` variable contains the string "San Francisco, CA", what will the correct statement in Question 14 assign to the `result` variable?

a. -1

b. 0

c. 11

d. 15

16. Which of the following statements replaces with the string "AB" the two characters located in subscripts 4 and 5 in a `string` variable named `code`?

a. `code.replace(2, 4, "AB");`

b. `code.replace(4, 2, "AB");`

c. `code.replace(4, 5, "AB");`

d. `replace(code, 4, "AB");`

17. Which of the following statements assigns five asterisks (*) to a `string` variable named `divider`?

a. `divider.assign(5, '*');`

b. `divider.assign(5, "*");`

c. `divider.assign('*', 5);`

d. `assign(divider, '*', 5);`

18. Which of the following statements concatenates the contents of a `string` variable named `city`, a comma, a space, and the contents of a `string` variable named `state` and then assigns the result to a `string` variable named `cityState`?

a. `cityState = "city" + ", " + "state";`

b. `cityState = city + ", " + state;`

c. `cityState = city & ", " & state;`

d. `cityState = "city, + state";`

19. Which of the following statements assigns the fifth character in the `word` variable to the `letter` variable?

    a. `letter = word.substr(4);`

    b. `letter = word.substr(4, 1);`

    c. `letter = word(5).substring;`

    d. `letter = substring(word, 5);`

20. Which of the following tells the computer to consume the next 100 characters?

    a. `cin.ignore(100);`

    b. `cin.ignore('100');`

    c. `ignore(cin, 100);`

    d. none of the above

21. When processed, the _____ can consume the newline character.

    a. `>>` operator

    b. `<<` operator

    c. `getline` function

    d. both a and c

Exercises



Pencil and Paper

TRY THIS

1. Write a C++ statement that assigns the number of characters contained in the `message` variable to an `int` variable named `numChars`. (The answers to TRY THIS Exercises are located at the end of the chapter.)

TRY THIS

2. Write a C++ statement that uses the `erase` function to remove the first two characters from the `message` variable. (The answers to TRY THIS Exercises are located at the end of the chapter.)

MODIFY THIS

3. Rewrite the code from Pencil and Paper Exercise 2 using the `replace` function.

INTRODUCTORY

4. Write a C++ statement that replaces with the letter "B" the first character in a `string` variable named `code`.

INTRODUCTORY

5. Write a C++ statement that assigns the first four characters in a `string` variable named `address` to a `string` variable named `streetNum`.

INTRODUCTORY

6. The `part` variable contains the string "ABCD34G". Write a C++ statement that assigns the 34 in the `part` variable to a `string` variable named `code`.

7. Write a C++ statement to change the contents of the `word` variable from "mend" to "amend". INTRODUCTORY

8. Write a C++ statement to change the contents of the `word` variable from "mouse" to "mouth". INTRODUCTORY

9. The `amount` variable contains the string "3,123,560". Write the C++ code to remove the commas from the contents of the variable. INTERMEDIATE

10. Write the C++ code that uses the `substr` function to determine whether the string stored in the `rate` variable ends with the percent sign. If it does, the code should use the `replace` function to remove the percent sign from the variable's contents. INTERMEDIATE

11. Write the C++ code to determine whether the `address` variable contains the street name "Grove Street". Begin the search with the fifth character in the `address` variable and assign the result to an `int` variable named `subNum` variable. INTERMEDIATE

12. Write a C++ statement that searches for the period in a `string` variable named `amount` and then assigns the location of the period to an `int` variable named `location`. Begin the search with the first character in the `amount` variable. INTERMEDIATE

13. The `total` and `dollars` variables are `string` variables. Write the C++ code that uses the `assign` function to assign 10 asterisks to the `total` variable. The code then should concatenate the contents of the `total` variable and the contents of the `dollars` variable and then assign the resulting string to the `total` variable. ADVANCED

14. A `string` variable named `amount` contains a string that has zero or more commas. Write the C++ code to count the number of commas in the string. Assign the result to an `int` variable named `numCommas`. ADVANCED

15. Correct the following statement, which should change the contents of the `day` variable from "731" to "7/31": `day = day.insert(2, "/");`. SWAT THE BUGS



Computer

16. If necessary, create a new project named TryThis16 Project. Enter the C++ instructions from Figure 13-22 into a source file named TryThis16.cpp. Change the filename in the first comment to TryThis16.cpp. Save and then run the program. Test the program using the data shown in Figure 13-23 in the chapter. Now, modify the program so the user enters the last name followed by a comma, a space, and the first name. The program should display the first name followed by a space and the last name. Be sure to modify the comments that document the program's purpose. Save and then run the program. Test the program appropriately. (The answers to TRY THIS Exercises are located at the end of the chapter.) TRY THIS

TRY THIS

17. If necessary, create a new project named TryThis17 Project. Also create a new source file named TryThis17.cpp. The program should allow the user to enter a string that represents a date. The date should be entered in the following format: mm/yy. Verify that the user entered exactly five characters and that the third character is the slash character (/). If the user did not enter the required number of characters, or if the third character is not a slash, display an appropriate message. Otherwise, the program should display the date in the following format: mm/20yy. Use a sentinel value to end the program. Save and then run the program. Test the program by entering the following dates: 6/08, 12/09, 05/10, and 123/4. (The answers to TRY THIS Exercises are located at the end of the chapter.)

MODIFY THIS

18. In this exercise, you modify the program from TRY THIS Exercise 17. If necessary, create a new project named ModifyThis18 Project. Copy the instructions from the TryThis17.cpp file into a source file named ModifyThis18.cpp. Change the filename in the first comment to ModifyThis18.cpp. Modify the program so that it allows the user to enter the date in the following format: mm/dd/yy. Verify that the user entered exactly eight characters and that the third and sixth characters are slashes (/). If the user did not enter the required number of characters, or if the third and sixth characters are not slashes, display an appropriate message. Otherwise, the program should display the date in the following format: mm/dd/20yy. Save and then run the program. Test the program appropriately.

MODIFY THIS

19. In this exercise, you modify the annual income program from the chapter. If necessary, create a new project named ModifyThis19 Project. Enter the C++ instructions from Figure 13-26 into a new source file named ModifyThis19.cpp. Change the filename in the first comment. Save and then run the program. Test the program using the data shown in Figure 13-27 in the chapter. Now, modify the program so that it verifies that the other characters in the annual income are numbers. Display an appropriate message if a non-numeric character is found; otherwise, display the annual income. Save and then run the program. Test the program appropriately.

INTRODUCTORY

20. If necessary, create a new project named Introductory20 Project. Also create a new source file named Introductory20.cpp. Write a program that displays the appropriate shipping charge based on the ZIP code entered by the user. To be valid, the ZIP code must contain exactly five digits and the first three digits must be either "605" or "606". The shipping charge for "605" ZIP codes is $25. The shipping charge for "606" ZIP codes is $30. Display an appropriate message if the ZIP code is invalid. Use a sentinel value to end the program. Save and then run the program. Test the program using the following ZIP codes: 60677, 60511, 60344, and 7130.

INTRODUCTORY

21. If necessary, create a new project named Introductory21 Project. Also create a new source file named Introductory21.cpp. Write a program that allows the user to enter three separate strings: a city name, state name, and ZIP code. The program should use string concatenation to

display the city name followed by a comma, a space, the state name, two spaces, and the ZIP code. Use a sentinel value to end the program. Save and then run the program. Test the program.

22. If necessary, create a new project named Intermediate22 Project. Also create a new source file named Intermediate22.cpp. Write a program that displays the color of the item whose item number is entered by the user. All item numbers contain exactly seven characters. All items are available in four colors: blue, green, red, and white. The fourth character in the item number indicates the item's color, as follows: a B or b indicates Blue, a G or g indicates Green, an R or r indicates Red, and a W or w indicates White. If the item number does not contain exactly seven characters, or if the fourth character is not one of the valid color characters, the program should display an appropriate message. Use a sentinel value to end the program. Save and then run the program. Test the program using the following item numbers: 123B567, 34AG123, 111r222, 111w222, 123, 1234567, and 111k456. INTERMEDIATE

23. In this exercise, you modify the Social Security number program from the chapter. If necessary, create a new project named Intermediate23 Project. Enter the instructions from Figure 13-32 into a source file named Intermediate23.cpp. Change the filename in the first comment. Before inserting the missing hyphens, verify that the nine characters entered by the user are numeric. Save and then run and test the program. INTERMEDIATE

24. If necessary, create a new project named Intermediate24 Project. Also create a new source file named Intermediate24.cpp. Write a program that accepts a string of characters from the user. The program should display the characters in reverse order. In other words, if the user enters the string "Programming", the program should display "gnimmargorP". Save and then run and test the program. INTERMEDIATE

25. If necessary, create a new project named Intermediate25 Project. Also create a new source file named Intermediate25.cpp. Write a program that allows the user to enter a part number that consists of four or five characters. The second and third characters represent the delivery method, as follows: "MS" represents "Mail – Standard", "MP" represents "Mail – Priority", "FS" represents "FedEx – Standard", "FO" represents "FedEx – Overnight", and "UP" represents "UPS". Display an appropriate message when the part number does not contain either four or five characters. Also display an appropriate message when the second and third characters are not one of the delivery methods. If the part number is valid, the program should display the delivery method. Use a sentinel value to end the program. Save and then run the program. Test the program using the following part numbers: 7MP6, 3fs5, 2UP7, 7mS89, 9FO8, 9fo89, 8ko89, and 1234MS. INTERMEDIATE

INTERMEDIATE

26. In this exercise, you modify the program from Lab 13-2. If necessary, create a new project named Intermediate26 Project. Also create a new source file named Intermediate26.cpp. Copy the C++ instructions from the Lab13-2.cpp file into the Intermediate26.cpp file. Change the filename in the first comment. Modify the program so that it displays a message indicating the number of incorrect guesses remaining. Display the message each time the user enters an incorrect guess.

ADVANCED

27. If necessary, create a new project named Advanced27 Project. Also create a new source file named Advanced27.cpp. Write a program that determines whether the user entered an item number in the required format: three digits, a hyphen, and two digits. Display an appropriate message indicating whether the format is correct. Use a sentinel value to end the program. Save and then run the program.

ADVANCED

28. Follow the instructions for starting C++ and opening the Advanced28.cpp file. The file is contained in either the Cpp6\Chap13\Advanced28 Project folder or the Cpp6\Chap13 folder. The program assigns the letters of the alphabet to a `string` variable named `letters`. It also prompts the user to enter a letter. Complete the program by entering instructions to perform the tasks listed in Figure 13-47. Save and then run the program. Test the program appropriately.

1. Generate a random number that can be used to select one of the letters from the `letters` variable. Assign the letter to the `randomLetter` variable.
2. Verify that the user entered exactly one letter. If the user did not enter exactly one letter, display an appropriate error message.
3. If the user entered exactly one letter, compare the letter to the random letter. If the letter entered by the user is the same as the random letter, display the message "You guessed the correct letter." and then end the program. Otherwise, display messages indicating whether the correct letter comes before or after the letter entered by the user.
4. Allow the user to enter a letter until he or she guesses the random letter.

Figure 13-47

ADVANCED

29. In this exercise, you modify the program from ADVANCED Exercise 28. If necessary, create a new project named Advanced29 Project. Also create a new source file named Advanced29.cpp. Copy the C++ instructions from the Advanced28.cpp file into the Advanced29.cpp file. Research the C++ `compare` function. Modify the program to use the `compare` function. Save and then run the program. Test the program appropriately.

ADVANCED

30. In this exercise, you modify the program from Lab 13-2. If necessary, create a new project named Advanced30 Project. Also create a new source file named Advanced30.cpp. Copy the C++ instructions from the Lab13-2.cpp file into the Advanced30.cpp file. Change the filename in the first comment. Modify the program so that it keeps track of the letters guessed by the user. If the user enters a letter that he or

she has already entered, display an appropriate message and do not include the letter in the number of incorrect guesses. Save and then run the program. Test the program appropriately.

31. In this exercise, you modify the program from ADVANCED Exercise 30. If necessary, create a new project named Advanced31 Project. Also create a new source file named Advanced31.cpp. Copy the instructions from the Advanced30.cpp file into the Advanced31.cpp file. Change the filename in the first comment. Modify the program so that it displays the letters already entered by the user. Display the letters immediately before prompting the user to enter a letter. Save and then run the program. Test the program appropriately. ADVANCED

32. If necessary, create a new project named Advanced32 Project. Also create a new source file named Advanced32.cpp. Create a program that allows the user to enter a word. The program should display the word in pig latin form. The rules for converting a word into pig latin form are listed in Figure 13-48. Use a sentinel value to end the program. ADVANCED

1. When the word begins with a vowel (A, E, I, O, or U), add the string "-way" (a dash followed by the letters w, a, and y) to the end of the word. For example, the pig latin form of the word "ant" is "ant-way".
2. When the word does not begin with a vowel, first add a dash to the end of the word. Then continue moving the first character in the word to the end of the word until the first character is the letter A, E, I, O, U, or Y. Then add the string "ay" to the end of the word. For example, the pig latin form of the word "Chair" is "air-Chay".
3. When the word does not contain the letter A, E, I, O, U, or Y, add the string "-way" to the end of the word. For example, the pig latin form of "56" is "56-way".

Figure 13-48

33. Some credit card companies assign a special digit, called a check digit, to the end of each customer's credit card number. Many methods for creating the check digit have been developed. One very simple method is to append the second digit in the credit card number to the end of the number. For example, if the first four characters in the credit card number are 1357, you would append the number 3 to the end of the number, making the credit card number 13573. If necessary, create a new project named Advanced33 Project. Also create a new source file named Advanced33.cpp. Write a program that prompts the user to enter a five-digit credit card number, with the fifth digit being the check digit. Verify that the user entered exactly five numbers. If the user entered the required number of characters, verify that the last character is the check digit. Display appropriate messages indicating whether the credit card number is valid or invalid. Use a sentinel value to end the program. Save and then run the program. Test the program appropriately. ADVANCED

ADVANCED

34. If necessary, create a new project named Advanced34 Project. Also create a new source file named Advanced34.cpp. Create a program that allows the user to enter a password. The program then should create and display a new password using the rules listed in Figure 13-49. Use a sentinel value to end the program.

1. All vowels (A, E, I, O, and U) in the original password should be replaced with the letter X.
2. All numbers in the original password should be replaced with the letter Z.
3. All of the characters in the original password should be reversed.

Figure 13-49

SWAT THE BUGS

35. Follow the instructions for starting C++ and opening the SwatTheBugs35.cpp file. The file is contained in either the Cpp6\Chap13\SwatTheBugs35 Project folder or the Cpp6\Chap13 folder. Debug the program.

Answers to TRY THIS Exercises



Pencil and Paper

1. `numChars = message.length();`

2. `message.erase(0, 2);`



Computer

16. See Figure 13-50.

```
 1  //TryThis16.cpp
 2  //Displays the first name followed by a space
 3  //and the last name
 4  //Created/revised by <your name> on <current date>
 5
 6  #include <iostream>
 7  #include <string>
 8  using namespace std;
 9
10  int main()
11  {
12      //declare variables
13      string firstLast   = "";
14      string first       = "";
15      string last        = "";
16      int commaLocation = 0;
17
18      //get first and last name
19      cout << "Name (last, comma, space, first): ";
20      getline(cin, firstLast);
```

Figure 13-50 *(continues)*

(continued)

```
21      //locate comma, then pull out first and
22      //last names
23      commaLocation = firstLast.find(",", 0);
24      last = firstLast.substr(0, commaLocation);
25      first = firstLast.substr(commaLocation + 2);
26
27      //display rearranged name
28      cout << first << " " << last << endl;
29
30      system("pause");
31      return 0;
32  }   //end of main function
```

Figure 13-50

17. See Figure 13-51.

```
 1  //TryThis17.cpp
 2  //Displays a date using the format mm/20yy
 3  //Created/revised by <your name> on <current date>
 4  #include <iostream>
 5  #include <string>
 6  using namespace std;
 7
 8  int main()
 9  {
10      string date = "";
11
12      cout << "Enter date (mm/yy). Enter -1 to end. ";
13      getline(cin, date);
14
15      while (date != "-1")
16      {
17          if (date.length() != 5)
18               cout << "Invalid length" << endl << endl;
19          else
20               if (date.substr(2, 1) != "/")
21                    cout << "Invalid third character"
22                    << endl << endl;
23               else
24               {
25                    date.insert(3, "20");
26                    cout << date << endl << endl;
27               }    //end if
28          //end if
29
30          cout << "Enter date (mm/yy). Enter -1 to end. ";
31          getline(cin, date);
32      }   //end while
33
34      system("pause");
35      return 0;
36  }   //end of main function
```

Figure 13-51

CHAPTER 14

Sequential Access Files

After studying Chapter 14, you should be able to:

- Create file objects
- Open a sequential access file
- Determine whether a sequential access file was opened successfully
- Write data to a sequential access file
- Read data from a sequential access file
- Test for the end of a sequential access file
- Close a sequential access file

File Types

In addition to getting data from the keyboard and sending data to the computer screen, a program also can get data from and send data to a file on a disk. Getting data from a file is referred to as "reading the file," and sending data to a file is referred to as "writing to the file." Files to which data is written are called **output files**, because the files store the output produced by a program. Files that are read by the computer are called **input files**, because a program uses the data in the files as input. Most input and output files are composed of lines of text that are both read and written sequentially. In other words, they are read and written in consecutive order, one line at a time, beginning with the first line in the file and ending with the last line in the file. Such files are referred to as **sequential access files**, because of the manner in which the lines of text are accessed. They also are referred to as **text files**, because they store text. Examples of text stored in sequential access files include an employee list, a memo, and a sales report.

You also can create random access and binary access files in C++. The data stored in a random access file can be accessed in either consecutive or random order. The data in a binary access file can be accessed by its byte location in the file. Random access and binary access files are used less often in programs and, therefore, are not covered in this book.

The CD Collection Program

The CD (compact disc) collection program, which you code in this chapter, will use a sequential access file to store the names of CDs along with the names of the artists. Figure 14-1 shows the program's problem specification and IPO charts. In addition to the `main` function, the CD collection program will use two void functions named `saveCd` and `displayCds`. The `saveCd` function will get both the CD's name and the artist's name from the user at the keyboard. It then will save the user's entries in a sequential access file. The `displayCds` function will display the contents of the sequential access file on the computer screen. You will learn how to code this program in the remainder of the chapter.

Problem specification
Create a program that keeps track of a CD (compact disc) collection. The program should display a menu containing the following three options:

1 Enter CD information
2 Display CD information
3 End the program

If the user selects option 1, the program should call a function that prompts the user to enter the CD's name and the artist's name and then saves the user's entries in a sequential access file named cds.txt. If the user selects option 2, the program should call a function that displays the contents of the cds.txt file on the screen. The program should end only when the user selects option 3. If the cds.txt file cannot be opened, the program should display the "File could not be opened." message.

Figure 14-1 Problem specification and IPO charts for the CD collection program *(continues)*

(continued)

`main` function

Input	Processing	Output
menu option	Processing items: none	none

```
Algorithm:
repeat
    display menu
    get menu option
    If (menu option is 1)
        call the saveCd function
    else
        if (menu option is 2)
            call the displayCds function
        end if
    end if
end repeat while (menu option is not 3)
```

`saveCd` function

Input	Processing	Output
CD name artist name	Processing items: none	cds.txt file (sequential access)

```
Algorithm:
1. open the cds.txt file for append
2. if (the cds.txt file was opened successfully)
        enter the CD name
        repeat while (the CD name is not "-1")
            enter the artist name
            write the CD name and artist name to the cds.txt file
            get another CD name
        end repeat
        close the cds.txt file
    else
        display the "File could not be opened" message
    end if
```

`displayCds` function

Input	Processing	Output
cds.txt file (sequential access)	Processing items: none	CD name artist name

```
Algorithm:
1. open the cds.txt file for input
2. if (the cds.txt file was opened successfully)
        read the CD name and artist name from the cds.txt file
        repeat while (it's not the end of the cds.txt file)
            display the CD name and artist name
            read the CD name and artist name from the cds.txt file
        end repeat
        close the cds.txt file
    else
        display the "File could not be opened" message
    end if
```

Figure 14-1 Problem specification and IPO charts for the CD collection program

Creating File Objects

In previous chapters, you used stream objects to perform standard input and output operations in a program. The standard input stream object (`cin`) refers to the computer keyboard, and the standard output stream object (`cout`) refers to the computer screen. As you already know, a program that uses the `cin` and `cout` objects must contain the `#include <iostream>` directive, which tells the compiler to include the contents of the iostream file in the program. The iostream file contains the definitions of the `istream` and `ostream` classes from which the `cin` and `cout` objects, respectively, are created. You do not have to create the `cin` and `cout` objects in a program, because C++ creates the objects in the iostream file for you. Objects also are used to perform file input and output operations in C++. However, unlike the standard `cin` and `cout` objects, the input and output file objects must be created by the programmer. To create a file object in a program, the program must contain the `#include <fstream>` directive, which tells the compiler to include the contents of the fstream file in the program. The fstream file contains the definitions of the `ifstream` (input file stream) and `ofstream` (output file stream) classes, which allow you to create input and output file objects, respectively. Figure 14-2 shows the syntax for creating input file objects and the syntax for creating output file objects. In each syntax, *fileObject* is the name of the file object you want to create. Figure 14-2 also includes examples of creating file objects. The statements in Examples 1 and 2 create input file objects named `inFile` and `inEmploy`. The statements in Examples 3 and 4 create output file objects named `outFile` and `outSales`. Notice that the names of the input file objects in the examples begin with the two letters `in`, whereas the names of the output file objects begin with the three letters `out`. Although the C++ syntax does not require you to begin file object names with either `in` or `out`, using this naming convention helps to distinguish a program's input file objects from its output file objects.



All objects in C++ are created from a class and are referred to as an instance of the class. For example, a `cin` object is an instance of the `istream` class, whereas an input file object is an instance of the `ifstream` class.

HOW TO Create Input and Output File Objects

Syntax

To create an input file object: **ifstream** *fileObject*;
To create an output file object: **ofstream** *fileObject*;

notice the semicolon

Example 1

```
ifstream inFile;
```

creates an input file object named `inFile`

Example 2

```
ifstream inEmploy;
```

creates an input file object named `inEmploy`

Example 3

```
ofstream outFile;
```

creates an output file object named `outFile`

Example 4

```
ofstream outSales;
```

creates an output file object named `outSales`

Figure 14-2 How to create input and output file objects

Opening a Sequential Access File

You use a program's input and output file objects, along with the C++ `open` function, to open actual files on your computer's disk. Figure 14-3 shows the `open` function's syntax and describes the modes most commonly used to open a sequential access file. In the syntax, *fileObject* is the name of either an existing `ifstream` file object or an existing `ofstream` file object, and *fileName* is the name of the file you want to open. Although it is not a requirement, many programmers use "txt" (short for "text") as the filename extension when naming sequential access files. The **open function** opens the file whose name is specified in the *fileName* argument and associates the file with the *fileObject*. When a subsequent statement in the program needs to refer to the file, it does so using the name of the *fileObject* rather than the *fileName* itself. The `open` function's *fileName* argument can be either a string literal constant or a `string` variable. In addition to the file's name, the *fileName* argument also can contain an optional path. If the *fileName* argument does not contain a path, the computer assumes that the file is located in the same folder as the program file. (See the second TIP on this page.) The optional *mode* argument in the syntax indicates how the file is to be opened. As Figure 14-3 indicates, you use the `ios::in` mode to open a file for input, which allows the computer to read the data stored in the file. The `ios::out` and `ios::app` modes are used to open output files. Both of these modes allow the computer to write data to the file. You use the `ios::app` (`app` stands for *append*) mode when you want to add data to the end of an existing file. If the file does not exist, the computer creates the file for you. You use the `ios::out` mode to open a new, empty file for output. If the file already exists, the computer erases the contents of the file before writing any data to it. The two colons (`::`) in each mode are called the scope resolution operator and indicate that the keywords `in`, `out`, and `app` are defined in the `ios` class. Also included in Figure 14-3 are examples of statements that open sequential access files. You can use either of the statements in Example 1 to open the payroll.txt file for input. Because the *fileName* argument in both statements does not contain a path, the computer will look for the payroll.txt file in the same location as the program file. Notice that the *mode* argument is omitted in the second statement in Example 1. Because all files associated with an `ifstream` file object are opened automatically for input, it is not necessary to specify `ios::in` when opening an input file. Unlike files associated with an `ifstream` object, files associated with an `ofstream` object are opened automatically for output. In other words, `ios::out` is the default mode when opening output files. This explains why you can use either of the statements in Example 2 to open the employ.txt file for output. Here too, because the *fileName* argument in both statements does not contain a path, the computer will look for the files in the same location as the program file. In cases for which the program needs to add data to the end of the existing data stored in an output file, you need to specify the `ios::app` mode in the `open` function, as shown in Example 3. In this case, the `outSales.open("F:\Cpp6\Chap14\sales.txt", ios::app);` statement tells the computer to open the sales.txt file, which is located in the Cpp6\Chap14 folder on the F drive, for append.



The `open` function is defined in the `ifstream` and `ofstream` classes and is referred to as a class member function.



In most cases, the program file refers to the .exe file. However, when running a program from the Microsoft Visual C++ IDE, the program file refers to the .cpp file.

HOW TO Open a Sequential Access File

Syntax

fileObject.**open(***fileName*[, *mode*]**);**

mode	Description
`ios::in`	Used with an `ifstream` object. Opens the file for input, which allows the computer to read the file's contents. This is the default mode for input files.
`ios::out`	Used with an `ofstream` object. Opens the file for output, which creates a new, empty file to which data can be written. If the file already exists, the computer erases the file's contents before the new data is written to it. This is the default mode for output files.
`ios::app`	Used with an `ofstream` object. Opens the file for append, which allows the computer to write new data to the end of the existing data in the file. If the file does not exist, the computer creates the file before writing any data to it.

Example 1

```
inFile.open("payroll.txt", ios::in);
```

or

```
inFile.open("payroll.txt");
```

opens the payroll.txt file for input

Example 2

```
outFile.open("employ.txt", ios::out);
```

or

```
outFile.open("employ.txt");
```

opens the employ.txt file for output

Example 3

```
outSales.open("F:\Cpp6\Chap14\sales.txt", ios::app);
```

opens the sales.txt file for append

Figure 14-3 How to open a sequential access file

The computer uses a file pointer to keep track of the next character either to read from or write to a file. When you open a file for input, the computer positions the file pointer at the beginning of the file, immediately before the first character. When you open a file for output, the computer also positions the file pointer at the beginning of the file, but recall that the file is empty. (As you learned earlier, when you open a file for output, the computer either creates a new, empty file or erases the contents of an existing file.) However, when you open a file for append, the computer positions the file pointer immediately after the last character in the file. Figure 14-4 illustrates the position of the file pointer when files are opened for input, output, and append.

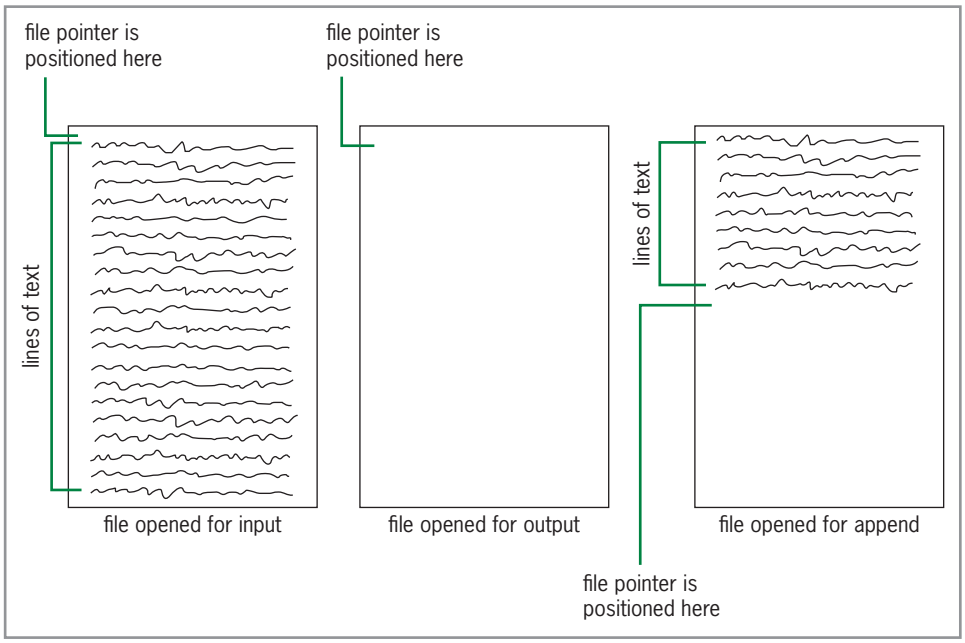

file pointer is positioned here
file pointer is positioned here
lines of text
lines of text
file opened for input
file opened for output
file opened for append
file pointer is positioned here

Figure 14-4 Position of the file pointer when files are opened for input, output, and append

The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 14-1

1. To create an input file in a program, the program must contain the _____ directive.
   a. `#include <filestream>`
   b. `#include <fstream>`
   c. `#include <instream>`
   d. `#include <iofilestream>`

2. Which mode tells the computer to open a file for input?
   a. `add::ios`
   b. `in::file`
   c. `ios::app`
   d. `ios::in`

3. Write the C++ statement to create an output file object named `outAlbums`.

4. Which of the following statements uses the `outAlbums` file object from Question 3 to open an output file named mine.txt? New information should be written following the existing information in the file.

   a. `outAlbums.open("mine.txt", ios::in);`

   b. `outAlbums.open("mine.txt", ios::out);`

   c. `outAlbums.open("mine.txt", ios::app);`

   d. `outAlbums.open("mine.txt", ios::add);`

Determining Whether a File Was Opened Successfully

Keep in mind that it is possible for the `open` function to fail when attempting to open a file. For example, the `open` function will not be able to create an output file when either the path specified in the *fileName* argument does not exist or the disk is full. It also will not be able to open an input file that does not exist or one that you don't have permission to open. After using the `open` function to open a file, you should use the `is_open` function to determine whether the file was opened successfully. The **is_open function** returns the Boolean value true if the `open` function was able to open the file; otherwise, it returns the Boolean value false. Figure 14-5 shows the `is_open` function's syntax. In the syntax, *fileObject* is the name of an existing file object in the program. Most times, you will use the `is_open` function in an `if` statement's condition, as shown in the examples in Figure 14-5. (For clarity, an appropriate `open` function is included in each example.) You can use either of the conditions shown in Example 1 to determine whether the `open` function was able to open the file associated with the `inFile` object. The first condition in Example 1, `inFile.is_open() == true`, compares the `is_open` function's return value to the Boolean value true. If the condition evaluates to true, it means that the `open` function was successful in opening the file. If the condition evaluates to false, it means that the `open` function was not able to open the file. As the second condition in Example 1 shows, you can omit the `== true` text from the condition and use `inFile.is_open()` instead. Unlike the conditions in Example 1, the conditions in Example 2 determine whether the `open` function failed to open the file associated with the `outFile` object. The `outFile.is_open() == false` condition compares the `is_open` function's return value to the Boolean value false. In this case, you can omit the `== false` text by preceding the condition with an exclamation point (!), as shown in the second condition in Example 2. The **!** is the **Not logical operator** in C++, and its purpose is to reverse the truth-value of the condition. In other words, if the value of `outFile.is_open()` is true, then the value of `!outFile.is_open()` is false. Likewise, if the value of `outFile.is_open()` is false, then the value of `!outFile.is_open()` is true.



The `is_open` function is a class member function in the `ifstream` and `ofstream` classes.

HOW TO Determine the Success of the `open` Function

Syntax

fileObject`.is_open()`

Example 1

```
inFile.open("payroll.txt");

if (inFile.is_open() == true)
```

or

```
if (inFile.is_open())
```

determines whether the `open` function succeeded in opening the file associated with the `inFile` object

Example 2

```
outFile.open("employ.txt");

if (outFile.is_open() == false)
```

or

```
if (!outFile.is_open())
```

determines whether the `open` function failed to open the file associated with the `outFile` object

Figure 14-5 How to determine the success of the `open` function

Writing Data to a Sequential Access File

Figure 14-6 shows the syntax for writing data to a sequential access file in C++. In the syntax, *fileObject* is the name of an existing `ofstream` object in the program, and *data* is the information you want written to the file. The figure also includes examples of using the syntax. The statement in Example 1 writes the string "XYZ Corporation" to the file associated with the `outFile` object and then advances the file pointer to the next line in the file. The first statement in Example 2 writes the string "Gross pay: " to the file associated with the `outFile` object, but it leaves the file pointer after the last character written—in this case, after the space character. The second statement in Example 2 writes the contents of the `gross` variable to the file and then advances the file pointer to the next line in the file. If the `gross` variable contains the number 450, the statements in Example 2 write "Gross pay: 450" (without the quotes) to the file before advancing the file pointer. In many programs, a sequential access file is used to store fields and records. A **field** is a single item of information about a person, place, or thing—such as a name, a salary, a Social Security number, or a price. A **record** is a collection of one or more related fields that contain all of the necessary data about a specific person, place, or thing. The college you are attending keeps a student record on you. Examples of fields contained in your student record include your Social Security number, name, address, phone number, credits earned, and grades earned. The place where you are employed also keeps a record on you. Your employee record contains your Social Security number, name, address, phone number, starting date, salary or hourly wage, and so on. To distinguish one



Fields and records are like columns and rows, respectively, in a table.

record from another in a sequential access file, programmers typically write each record on a separate line in the file. You do this by including the `endl` stream manipulator at the end of the statement that writes the record. The `outSales << custName << endl;` statement in Example 3, for instance, writes a record that contains one field (the name stored in the `custName` variable) to the file associated with the `outSales` object. The `endl` stream manipulator writes a newline character at the end of the record. As you learned in Chapter 4, the newline character represents the Enter key. In this case, the newline character advances the file pointer to the next line in the file. When writing a record that contains more than one field, programmers typically separate each field with a character literal constant. The character literal constant `'#'`, which is the number sign enclosed in single quotation marks, is commonly used as the separator character. The `'#'` character appears in the `outEmploy << name << '#' << salary << endl;` statement in Example 4. The statement writes a record that contains two fields: the name stored in the `name` variable and the salary amount stored in the `salary` variable. The statement writes the record on a separate line in the file, with the number sign separating the data in the `name` field from the data in the `salary` field.

HOW TO Write Data to a Sequential Access File

Syntax

fileObject **<<** *data*;

Example 1

```
outFile << "XYZ Corporation" << endl;
```

writes the string "XYZ Corporation" to the file associated with the `outFile` object and then advances the file pointer to the next line in the file

Example 2

```
outFile << "Gross pay: ";
outFile << gross << endl;
```

writes the string "Gross pay: " and the contents of the `gross` variable to the file associated with the `outFile` object and then advances the file pointer to the next line in the file

Example 3

```
outSales << custName << endl;
```

writes the contents of the `custName` variable to the file associated with the `outSales` object and then advances the file pointer to the next line in the file

Example 4

```
outEmploy << name << '#' << salary << endl;
```

writes the contents of the `name` variable, the number sign (#), and the contents of the `salary` variable to the file associated with the `outEmploy` object and then advances the file pointer to the next line in the file

Figure 14-6 How to write data to a sequential access file



You can use Notepad to create a text file. However, when saving the file, be sure to enclose the filename in quotation marks, like this: "employees.txt". You also can use Notepad to create a .cpp file, as long as you enclose the filename in quotation marks.

You can verify that the information was written correctly to a sequential access file by opening the file in a text editor, such as the text editor in Visual C++, Dev-C++, or Notepad. The instructions for opening a sequential access file depend on the text editor you are using. However, in most cases, you click File on the text editor's menu bar. If you are using either Microsoft Visual C++ or Dev-C++, you then display the Open File dialog box by pointing to Open and then clicking File (Visual C++) or by clicking Open Project or File (Dev-C++). If you are using Notepad, you display the Open dialog box by clicking Open on the File menu. Next, you click the name of the appropriate file in the Open File (or Open) dialog box and then click the Open button. Figure 14-7 shows a sequential access file named employees.txt opened in a text editor. Because the newline character is invisible, you will not see it when you open a sequential access file.

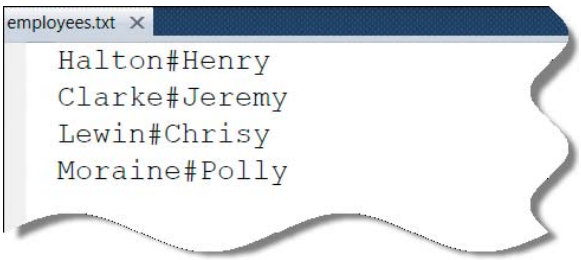


Figure 14-7 The employees.txt sequential access file opened in a text editor

Reading Information from a Sequential Access File

Figure 14-8 shows the syntax for reading numeric and `char` data from a sequential access file in C++, as well as the syntax for reading `string` data. The figure also includes examples of using each syntax. In each syntax, *fileObject* is the name of an existing `ifstream` object in the program. The *variableName* and *stringVariableName* arguments represent the name of the variable that will store the information read from the file. As Figure 14-8 indicates, you use the extraction operator (>>) to read `char` and numeric data from a file. The `inFile >> years;` and `inFile >> salary;` statements in Example 1, for instance, read numbers from the file associated with the `inFile` object and store the numbers in the `years` and `salary` variables. The `inAlphabet >> letter;` statement in Example 2 reads a character from the file associated with the `inAlphabet` object and stores the character in the `letter` variable. To read `string` data from a sequential access file, you use the `getline` function, which you learned about in Chapter 13. The `getline` function will continue to read characters from the file associated with the *fileObject* until it encounters the *delimiterCharacter*, which it consumes. Recall that when a function consumes a character, it means that the function reads and discards the character. If you omit the *delimiterCharacter* argument in the `getline` function, the default delimiter character is the newline character. The `getline(inEmploy, name);` statement in Example 3 uses the `getline` function to read a string from the file

associated with the `inEmploy` object. Because the `getline` function does not specify a *delimiterCharacter*, the function stops reading when it encounters the newline character. The function stores the string in the `name` variable and then consumes the newline character. The `getline(inEmploy, name, '#');` statement in Example 4 also reads a string from the file associated with the `inEmploy` object. In this case, however, the `getline` function's *delimiterCharacter* argument indicates that the string ends with the character immediately preceding the # character. After storing the string in the `name` variable, the `getline` function consumes the # character. The next statement in the example, `inEmploy >> salary;`, reads a number from the file and stores the number in the `salary` variable.

HOW TO Read Data from a Sequential Access File

Syntax

To read numeric and `char` data: *fileObject* **>>** *variableName*;
To read `string` data: **getline(***fileObject*, *stringVariableName*[, *delimiterCharacter*]**);**

Example 1

```
int years = 0;
double salary = 0.0;
inFile >> years;
inFile.ignore(1);
inFile >> salary;
```

reads a number from the file associated with the `inFile` object and stores the number in the `years` variable, then ignores (consumes) one character, and then reads the next number and stores it in the `salary` variable

Example 2

```
char letter = ' ';
inAlphabet >> letter;
```

reads a character from the file associated with the `inAlphabet` object and stores the character in the `letter` variable

Example 3

```
string name = "";
getline(inEmploy, name);
```

reads a string from the file associated with the `inEmploy` object and stores the string in the `name` variable

Example 4

```
string name = "";
double salary = 0.0;
getline(inEmploy, name, '#');
inEmploy >> salary;
```

reads a string from the file associated with the `inEmploy` object and stores the string in the `name` variable, then consumes the # character, and then reads a number from the file and stores the number in the `salary` variable

Figure 14-8 How to read data from a sequential access file



The answers to Mini-Quiz questions are located in Appendix A.

Mini-Quiz 14-2

1. What value does the `is_open` function return when the `open` function fails?

2. Which of the following statements writes the contents of the `quantity` variable to the inventory.txt file, which is associated with a file object named `outInv`?

   a. `inventory.txt << quantity << endl;`

   b. `ofstream << quantity << endl;`

   c. `outInv << quantity << endl;`

   d. `outInv >> quantity >> endl;`

3. Which of the following statements writes a record to the test.txt file? The file is associated with a file object named `outFile`. The record contains two scores, which are stored in the `score1` and `score2` variables.

   a. `test.txt << score1 << score2 << endl;`

   b. `ofstream << score1 << '#' << score2 << endl;`

   c. `outFile << score1 << score2 << endl;`

   d. `outFile << score1 << '#' << score2 << endl;`

4. Which of the following statements reads a record written by the statement from Question 2 and stores the record in the `number` variable? The inventory.txt file is associated with a file object named `inInv`.

   a. `ifstream >> number;`

   b. `inventory.txt >> number;`

   c. `inInv << number;`

   d. `inInv >> number;`

Testing for the End of a Sequential Access File

As you learned earlier, the computer uses a file pointer to keep track of the next character either to read from a file or write to a file. When a sequential access file is opened for input, the computer positions the file pointer before the first character in the file. Each time a character is read from the file, the file pointer is moved to the next character. When an entire line from the file is read, the computer moves the file pointer to the beginning of the next line in the file. Most times, a program will need to read each line contained in a sequential access file, one line at a time, beginning with the first line and ending with the last line. You can accomplish this task

using a loop along with the `eof` (end of file) function. The **eof function** determines whether the last character in a file has been read. In other words, it determines whether the file pointer is located after the last character in the file. If the file pointer is located at the end of the file, the `eof` function returns the Boolean value true; otherwise, it returns the Boolean value false. Figure 14-9 shows the function's syntax and includes examples of using the function. In the syntax, *fileObject* is the name of an existing `ifstream` object in the program. The condition in the `while` clause in Example 1 tells the computer to repeat the loop instructions as long as the end of the file has not been reached. You also can write the condition using the Not logical operator (`!`), as shown in Example 2.



The `eof` function is a class member function in the `ifstream` class.

HOW TO Test for the End of a Sequential Access File

Syntax

fileObject.**eof()**

Example 1

```
while (inFile.eof() == false)
```

tells the computer to repeat the loop instructions as long as (or while) the end of the file associated with the `inFile` object has not been reached

Example 2

```
while (!inFile.eof())
```

same as Example 1

Figure 14-9 How to test for the end of a sequential access file

Closing a Sequential Access File

To prevent the loss of data, you should use the **close function** to close a sequential access file as soon as the program is finished using it. The function's syntax is shown in Figure 14-10 along with examples of using the function. In the syntax, *fileObject* is the name of either an existing `ifstream` object or an existing `ofstream` object in the program. Notice that the `close` function does not require the name of the file you want to close. This is because the computer automatically closes the file whose name is associated with the *fileObject*. (Recall that the `open` function associates the file's name with the *fileObject* when the file is opened.) The `close` function in Example 1 closes the input file associated with the `inFile` object. The `close` function in Example 2 closes the output file associated with the `outFile` object. Because it is so easy to forget to close the files used in a program, you should enter the statement to close the file as soon as possible after entering the one that opens it.



The `close` function is a class member function in the `ifstream` and `ofstream` classes.

HOW TO Close a Sequential Access File

Syntax

fileObject.**close()**

Example 1

`inFile.close()`
closes the file associated with the `inFile` object

Example 2

`outFile.close()`
closes the file associated with the `outFile` object

Figure 14-10 How to close a sequential access file

Coding the CD Collection Program

Figure 14-11 shows the IPO chart information and C++ instructions for the CD collection program.

`main` function

IPO chart information	**C++ instructions**
Input	
menu option	`int menuOption = 0;`
Processing	
none	
Output	
none	
Algorithm	
repeat	`do`
	`{`
display menu	`cout << endl;`
	`cout << "1  Enter CD information"`
	`<< endl;`
	`cout << "2  Display CD information"`
	`<< endl;`
	`cout << "3  End the program" << endl;`
get menu option	`cout << "Enter menu option: ";`
	`cin >> menuOption;`
	`cin.ignore(100, '\n');`
	`cout << endl;`

Figure 14-11 IPO chart information and C++ instructions for the CD collection program *(continues)*

(continued)

if (menu option is 1)
  call the saveCd function
else
  if (menu option is 2)
  call the displayCds function
  end if
end if
end repeat while (menu option is not 3)
	if (menuOption == 1)
	  saveCd();
	else
	  if (menuOption == 2)
	  displayCds();
	  //end if
	//end if
	} while (menuOption != 3);

saveCd function	
IPO chart information	**C++ instructions**
Input	
CD name artist name	string cdName = ""; string artistName = "";
Processing	
none	
Output	
cds.txt file (sequential access)	ofstream outFile;
Algorithm	

1. open the cds.txt file for append
2. if (the cds.txt file was opened successfully)
  enter the CD name
  repeat while (the CD name is not "-1")
  enter the artist name
  write the CD name and artist name to the cds.txt file
  get another CD name
  end repeat
  close the cds.txt file
else
  display the "File could not be opened." message
end if
	outFile.open("cds.txt", ios::app);
	if (outFile.is_open())
	{
	  cout << "CD name (-1 to stop): ";
	  getline(cin, cdName);
	  while (cdName != "-1")
	  {
	  cout << "Artist's name: ";
	  getline(cin, artistName);
	  outFile << cdName << '#'
	  << artistName << endl;
	  cout << "CD name (-1 to stop): ";
	  getline(cin, cdName);
	  } //end while
	  outFile.close();
	}
	else
	  cout << "File could not be
	  opened." << endl;
	//end if

displayCds function	
IPO chart information	**C++ instructions**
Input	
cds.txt file (sequential access)	ifstream inFile;
Processing	
none	

Figure 14-11 IPO chart information and C++ instructions for the CD collection program *(continues)*

(continued)

Output

```
CD name                                  string cdName     = "";
artist name                              string artistName = "";
```

Algorithm

```
1. open the cds.txt file for input       inFile.open("cds.txt", ios::in);
2. if (the cds.txt file was opened       if (inFile.is_open())
   successfully)                         {
      read the CD name and artist           getline(inFile, cdName, '#');
      name from the cds.txt file            getline(inFile, artistName);

      repeat while (it's not the end        while (!inFile.eof())
      of the cds.txt file)                  {
         display the CD name and               cout << cdName << ", " <<
         artist name                           artistName << endl;

         read the CD name and artist           getline(inFile, cdName, '#');
         name from the cds.txt file            getline(inFile, artistName);
      end repeat                            }  //end while
      close the cds.txt file                inFile.close();
                                         }
   else                                  else
      display the "File could not be        cout << "File could not
      opened" message                       be opened." << endl;
   end if                                //end if
```

Figure 14-11 IPO chart information and C++ instructions for the CD collection program

Figure 14-12 shows the entire CD collection program, with the instructions pertaining to sequential access files shaded. Figure 14-13 shows a sample run of the program, and Figure 14-14 shows the cds.txt file opened in a text editor.

```
 1  //CD Collection.cpp - keeps track of a CD collection
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <string>
 6  #include <fstream>
 7  using namespace std;
 8
 9  //function prototypes
10  void saveCd();
11  void displayCds();
12
13  int main()
14  {
15       int menuOption = 0;
16
```

Figure 14-12 CD collection program *(continues)*

(continued)

```
17      do  //begin loop
18      {
19          //display menu and get option
20          cout << endl;
21          cout << "1  Enter CD information" << endl;
22          cout << "2  Display CD information" << endl;
23          cout << "3  End the program" << endl;
24          cout << "Enter menu option: ";
25          cin >> menuOption;
26          cin.ignore(100, '\n');
27          cout << endl;
28
29          //call appropriate function
30          //or display error message
31          if (menuOption == 1)
32               saveCd();
33          else
34               if (menuOption == 2)
35                    displayCds();
36               //end if
37          //end if
38      }   while (menuOption != 3);
39
40      system("pause");
41      return 0;
42  }   //end of main function
43
44  //*****function definitions*****
45  void saveCd()
46  {
47      //writes records to a sequential access file
48      string cdName     = "";
49      string artistName = "";
50
51      //create file object and open the file
52      ofstream outFile;
53      outFile.open("cds.txt", ios::app);
54
55      //determine whether the file was opened
56      if (outFile.is_open())
57      {
58          //get the CD name
59          cout << "CD name (-1 to stop): ";
60          getline(cin, cdName);
61          while (cdName != "-1")
62          {
63               //get the artist's name
64               cout << "Artist's name: ";
65               getline(cin, artistName);
66               //write the record
67               outFile << cdName << '#'
68                    << artistName << endl;
```

your C++ development tool may not require this statement

Figure 14-12 CD collection program *(continues)*

(continued)

```
69              //get another CD name
70              cout << "CD name (-1 to stop): ";
71              getline(cin, cdName);
72          } //end while
73
74          //close the file
75          outFile.close();
76      }
77      else
78          cout << "File could not be opened." << endl;
79      //end if
80  } //end of saveCd function
81
82  void displayCds()
83  {
84      //displays the records stored in the cds.txt file
85      string cdName     = "";
86      string artistName = "";
87
88      //create file object and open the file
89      ifstream inFile;
90      inFile.open("cds.txt", ios::in);
91
92      //determine whether the file was opened
93      if (inFile.is_open())
94      {
95          //read a record
96          getline(inFile, cdName, '#');
97          getline(inFile, artistName);
98
99          while (!inFile.eof())
100         {
101             //display the record
102             cout << cdName << ", " <<
103                  artistName << endl;
104             //read another record
105             getline(inFile, cdName, '#');
106             getline(inFile, artistName);
107         } //end while
108
109         //close the file
110         inFile.close();
111     }
112     else
113         cout << "File could not be opened." << endl;
114     //end if
115 }  //end of displayCds function
```

Figure 14-12 CD collection program

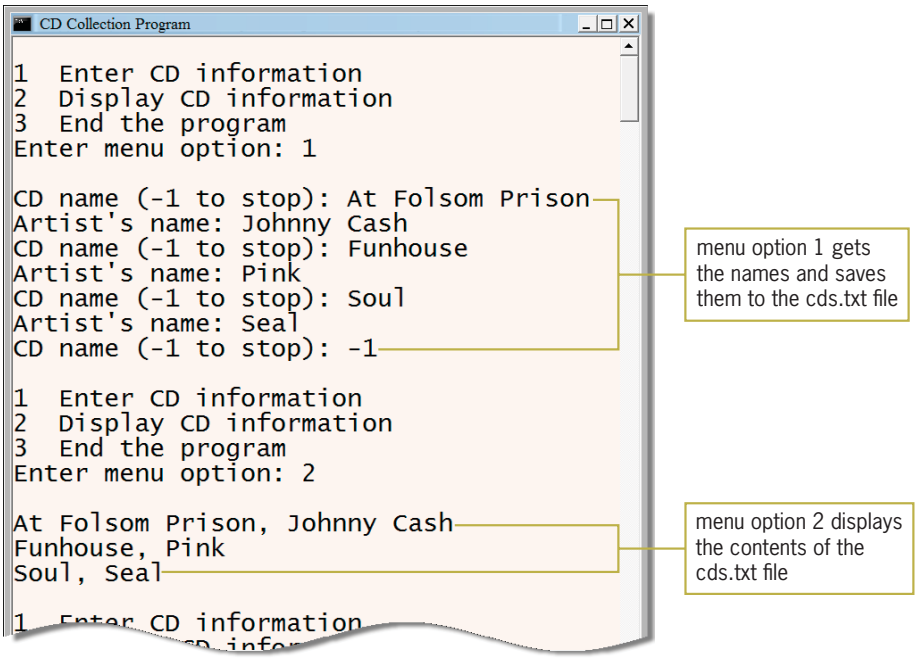

CD Collection Program
1 Enter CD information
2 Display CD information
3 End the program
Enter menu option: 1
CD name (-1 to stop): At Folsom Prison
Artist's name: Johnny Cash
CD name (-1 to stop): Funhouse
Artist's name: Pink
CD name (-1 to stop): Soul
Artist's name: Seal
CD name (-1 to stop): -1
1 Enter CD information
2 Display CD information
3 End the program
Enter menu option: 2
At Folsom Prison, Johnny Cash
Funhouse, Pink
Soul, Seal
menu option 1 gets the names and saves them to the cds.txt file
menu option 2 displays the contents of the cds.txt file

Figure 14-13 Sample run of the CD collection program

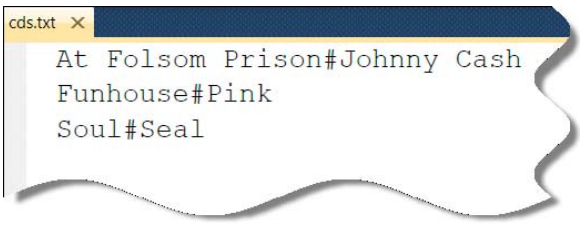


Figure 14-14 The cds.txt sequential access file opened in a text editor

Mini-Quiz 14-3



The answers to Mini-Quiz questions are located in Appendix A.

1. Which of the following `while` clauses tells the computer to continue reading the inventory.txt file until the end of the file is reached? The file object is named `inInv`.

   a. `while (inventory.txt.end())`

   b. `while (inInv.end())`

   c. `while (!inInv.eof())`

   d. `while (!inventory.txt.eof())`

2. What value does the `eof` function return when the file pointer is not at the end of the file?

3. Write the statement to close the inventory.txt file, which is associated with a file object named `outInv`.



The answers to the labs are located in Appendix A.



LAB 14-1 Stop and Analyze

Study the program shown in Figure 14-15 and then answer the questions.


```
1  //Lab14-1.cpp - saves movie titles and release
2  //years in a sequential access file
3  //Created/revised by <your name> on <current date>
4
5  #include <iostream>
6  #include <string>
7  #include <fstream>
8
9  using namespace std;
10
11 int main()
12 {
13     string title = "";
14     string year  = "";
15
16     //create file object and open the file
17     ofstream outFile;
18     outFile.open("movies.txt", ios::out);
19
20     //determine whether the file is open
21     if (outFile.is_open())
22     {
23         //get movie title
24         cout << "Movie title (-1 to stop): ";
25         getline(cin, title);
26
27         while (title != "-1")
28         {
29              //get the release year
30              cout << "Year released: ";
31              getline(cin, year);
32
33              //write the record to the file
34              outFile << title << '#' << year << endl;
35
36              //get another movie title
37              cout << "Movie title (-1 to stop): ";
38              getline(cin, title);
39         }    //end while
40
41         //close the file
42         outFile.close();
43     }
44     else
45         cout << "The file could not be opened." << endl;
46     //end if
47
48     system("pause");
49     return 0;
50 }   //end of main function
```

your C++ development tool may not require this statement

Figure 14-15 Code for Lab 14-1

QUESTIONS

1. Why are the instructions in Lines 5, 6, and 7 necessary?
2. The program writes records to a sequential access file. How many fields are in each record and what are they?
3. Suppose you run the program twice, entering three records the first time and two records the second time. If you open the movies.txt file in a text editor, how many records will the file contain and why?
4. How can you modify the program so that the existing records in the movies.txt file are not erased when the program is run?
5. What is another way of writing the `if` clause in Line 21?
6. What is the purpose of the # character in Line 34?
7. Why is the statement in Line 42 necessary?
8. Follow the instructions for starting C++ and opening the Lab14-1.cpp file. The file is contained in either the Cpp6\Chap14\Lab14-1 Project folder or the Cpp6\Chap14 folder. Run the program. When you are prompted to enter a movie title, type Titanic and press Enter. When you are prompted to enter the release year, type 1997 and press Enter. Next, enter The Dark Knight as the movie title and 2008 as the release year. Finally, enter -1 as the movie title.
9. Use a text editor to open the movies.txt file. The file contains two records. Close the movies.txt file.
10. Run the program again. Enter Shrek 2 as the movie title and 2004 as the release year, and then enter -1 as the movie title. Use a text editor to open the movies.txt file. The file contains one record. Close the movies.txt file.
11. Modify the program so that the existing records in the movies.txt file are not erased when the program is run.
12. Save and then run the program. Enter the following movie titles and release years:

Titanic	1997
The Dark Knight	2008
Star Wars: Episode IV – A New Hope	1977
E.T. the Extra-Terrestrial	1982
-1	

13. Use a text editor to open the movies.txt file. The file contains five records. Close the movies.txt file.



LAB 14-2 Plan and Create

In this lab, you will plan and create an algorithm for Cheryl Liu, the owner of a candy shop named Sweets-4-You. The problem specification is shown in Figure 14-16.

> Cheryl Liu is the owner of a candy shop named Sweets-4-You. She wants a program that uses a function to display the following menu:
>
> Menu Options
> 1 Add Records
> 2 Display Total Sales
> 3 Exit the program
>
> If Cheryl selects option 1, the program should call a function that prompts Cheryl to enter each salesperson's name and sales amount. The function should save Cheryl's entries in a sequential access file named sales.txt. If Cheryl selects option 2, the program should call a function that calculates and displays the total of the sales amounts stored in the sales.txt file. The program should end only when Cheryl selects option 3.

Figure 14-16 Problem specification for Lab 14-2

The Sweets-4-You program will use four functions: `main`, `getChoice`, `addRecords`, and `displayTotal`. Figure 14-17 shows the IPO chart information and C++ instructions for the `main` and `getChoice` functions. The `main` function begins by declaring and initializing an `int` variable named `choice`. It then calls the `getChoice` function to display the menu, which contains three options. After displaying the menu, the `getChoice` function prompts Cheryl to enter her choice of menu options: 1 to add records, 2 to display the total sales, or 3 to exit the program. The `getChoice` function returns Cheryl's response to the `main` function, which assigns the value to the `choice` variable. The next instruction in the `main` function is a selection structure that compares the contents of the `choice` variable with the number 1. If the `choice` variable contains the number 1, the `main` function calls the `addRecords` function to add one or more records to the sequential access file. When the `addRecords` function ends, the `main` function calls the `getChoice` function to display the menu again and get another choice from the user. If the `choice` variable does not contain the number 1, the nested selection structure in the `main` function compares the contents of the `choice` variable with the number 2. If the `choice` variable contains the number 2, the `main` function calls the `displayTotal` function to total the sales amounts stored in the sequential access file and then display the total on the screen. When the `displayTotal` function ends, the `main` function calls the `getChoice` function once again. If the `choice` variable contains the number 3, the program ends. If the `choice` variable contains a value other than 1, 2, or 3, the `main` function simply calls the `getChoice` function.

main function

IPO chart information	C++ instructions
Input	
menu choice	`int choice = 0;`
Processing	
none	
Output	
none	
Algorithm	
repeat	`do`
	`{`
call the getChoice function to display the menu and get the menu choice	`choice = getChoice();`
if (menu choice is 1)	`if (choice == 1)`
call the addRecords function	`addRecords();`
else	`else`
if (menu choice is 2)	`if (choice == 2)`
call the displayTotal function	`displayTotal();`
end if	`//end if`
end if	`//end if`
end repeat while (menu choice is not 3)	`} while (choice != 3);`

getChoice function

IPO chart information	C++ instructions
Input	
none	
Processing	
none	
Output	
menu choice	`int menuChoice = 0;`
Algorithm	

1. display menu	`cout << endl << "Menu Options" << endl;`
	`cout << "1 Add Records" << endl;`
	`cout << "2 Display Total Sales"`
	`<< endl;`
	`cout << "3 Exit the program" << endl;`

2. get menu choice	`cout << "Choice (1, 2,`
	`or 3)? ";`
	`cin >> menuChoice;`
	`cin.ignore(100, '\n');`
	`cout << endl;`

3. return menu choice	`return menuChoice;`

Figure 14-17 IPO chart information and C++ instructions for the `main` and `getChoice` functions

Figure 14-18 shows the IPO chart information and C++ instructions for the `addRecords` function. The function creates an output file object named `outFile` and then uses the object, along with the `open` function, to open the sales.txt file for append. The condition in the `if` clause determines whether the sales.txt file was opened successfully. If the condition evaluates to false, it means that the `open` function failed to open the file. In that case, the `addRecords` function displays an appropriate error message and then the function ends. If the condition evaluates to true, on the other hand, it means that the `open` function was able to open the sales.txt file. As a result, the instructions in the `if` statement's true path are processed. The first two statements in the true path prompt the user to enter the salesperson's name and then store the user's response in the `name` variable. The `while` clause in the true path is processed next. The `while` clause indicates that the loop body instructions should be repeated as long as the `name` variable does not contain either the letter X or the letter x. The first two statements in the loop body prompt the user to enter the sales amount and then store the user's response in the `sales` variable. The `cin.ignore(100,'\n');` statement instructs the computer to consume the newline character that remains in the `cin` object after the sales amount is entered. The `outFile << name << '#' << sales << endl;` statement then writes a record, followed by a newline character, to the file. The record contains the contents of the `name` variable, the # character, and the contents of the `sales` variable. The last two statements in the loop body prompt the user to enter another salesperson's name and then store the user's response in the `name` variable. The loop will end when the `name` variable contains either the string "X" or the string "x". When the loop ends, the `outFile.close();` statement closes the sales.txt file before the `addRecords` function ends.

addRecords function

IPO chart information	**C++ instructions**
Input	
salesperson's name	`string name = "";`
sales amount	`int sales = 0;`
Processing	
none	
Output	
sales.txt file (sequential access)	`ofstream outFile;`
Algorithm	
1. *open the sales.txt file for append*	`outFile.open("sales.txt", ios::app);`
2. *if (the sales.txt file was opened successfully)*	`if (outFile.is_open())`
enter the salesperson's name	`{`
	`cout << "Salesperson's name (X to stop): ";`
	`getline(cin, name);`

Figure 14-18 IPO chart information and C++ instructions for the `addRecords` function *(continues)*

(continued)

```
   repeat while (the salesperson's          while (name != "X"
   name is not "X" or "x")                  && name != "x")
                                            {
      enter the sales amount                   cout << "Sales: ";
                                               cin >> sales;
                                               cin.ignore(100, '\n');

      write the salesperson's name             outFile << name << '#'
      and sales amount to the                  << sales << endl;
      sales.txt file

      get another salesperson's name           cout << "Salesperson's name
                                               (X to stop): ";
                                               getline(cin, name);
   end repeat                               }  //end while
   close the sales.txt file                 outFile.close();
                                         }
else                                     else
   display the "File could not              cout << "File could not be
   be opened." message                      opened." << endl;
end if                                    //end if
```

Figure 14-18 IPO chart information and C++ instructions for the `addRecords` function

Finally, Figure 14-19 shows the IPO chart information and C++ instructions for the `displayTotal` function. The function creates an input file object named `inFile` and then uses the object, along with the `open` function, to open the sales.txt file for input. The condition in the `if` clause determines whether the sales.txt file was opened successfully. If the condition evaluates to false, the `displayTotal` function displays an appropriate error message and then the function ends. If the condition evaluates to true, on the other hand, the instructions in the `if` statement's true path are processed. The instructions in the true path read a record from the sales.txt file, assigning the name to the `name` variable and assigning the sales to the `sales` variable. The `while` clause in the true path tells the computer to repeat the loop body instructions as long as the file pointer is not at the end of the file. The first statement in the loop body adds the sales amount to the accumulator variable, which is named `total`. The remaining instructions in the loop body read another record from the file. When the loop ends, which occurs when the file pointer is at the end of the sales.txt file, the last two statements in the `if` statement's true path close the sales.txt file and then display the total sales amount on the screen. After displaying the total sales amount, the `displayTotal` function ends.

displayTotal function

IPO chart information	**C++ instructions**
Input	
sales.txt file (sequential access)	`ifstream inFile;`
Processing	
salesperson's name	`string name = "";`
sales amount	`int sales = 0;`
Output	
total sales amount (accumulator)	`int total = 0;`
Algorithm	
1. open the sales.txt file for input	`inFile.open("sales.txt");`
2. if (the sales.txt file was opened successfully)	`if (inFile.is_open())`
	`{`

read the salesperson's name and sales amount from the sales.txt file	`getline(inFile, name, '#');`
	`inFile >> sales;`
	`inFile.ignore();`

repeat while (it's not the end of the sales.txt file)	`while (!inFile.eof())`
	`{`
add the sales amount to the total sales amount	`total += sales;`

read the salesperson's name and sales amount from the sales.txt file	`getline(inFile, name, '#');`
	`inFile >> sales;`
	`inFile.ignore();`

end repeat	`} //end while`
close the sales.txt file	`inFile.close();`
display the total sales amount	`cout << "Total sales $"` `<< total << endl << endl;`
	`}`
else	`else`
display the "File could not be opened." message	`cout << "File could not be opened." << endl;`
end if	`//end if`

Figure 14-19 IPO chart information and C++ instructions for the `displayTotal` function

Figure 14-20 shows the code for the entire CD collection program, and Figure 14-21 shows a sample run of the program.

```
 1  //Lab14-2.cpp - saves records to a sequential access
 2  //file and also calculates and displays the total
 3  //of the sales amounts stored in the file
 4  //Created/revised by <your name> on <current date>
 5
 6  #include <iostream>
 7  #include <string>
 8  #include <fstream>
 9  using namespace std;
10
11  //function prototypes
12  int getChoice();
13  void addRecords();
14  void displayTotal();
15
16  int main()
17  {
18      int choice = 0;
19      do
20      {
21          //get user's menu choice
22          choice = getChoice();
23          if (choice == 1)
24               addRecords();
25          else
26               if (choice == 2)
27                    displayTotal();
28                //end if
29          //end if
30      }   while (choice != 3);
31
32          system("pause");
33          return 0;
34  }   //end of main function
35
36  //*****function definitions*****
37  int getChoice()
38  {
39      //displays menu and returns choice
40
41      int menuChoice = 0;
42      cout << endl << "Menu Options" << endl;
43      cout << "1  Add Records" << endl;
44      cout << "2  Display Total Sales" << endl;
45      cout << "3  Exit the program" << endl;
46      cout << "Choice (1, 2, or 3)? ";
47      cin >> menuChoice;
48      cin.ignore(100, '\n');
49      cout << endl;
50      return menuChoice;
51  }   //end of getChoice function
52
53  void addRecords()
54  {
55      //saves records to a sequential access file
56
```

your C++ development tool may not require this statement

Figure 14-20 CD collection program *(continues)*

(continued)

```
57      string name = "";
58      int sales   = 0;
59      ofstream outFile;
60
61      //open file for append
62      outFile.open("sales.txt", ios::app);
63
64      //if the open was successful, get the
65      //salesperson's name and sales amount and
66      //then write the information to the file;
67      //otherwise, display an error message
68      if (outFile.is_open())
69      {
70          cout << "Salesperson's name (X to stop): ";
71          getline(cin, name);
72
73          while (name != "X" && name != "x")
74          {
75               cout << "Sales: ";
76               cin >> sales;
77               cin.ignore(100, '\n');
78
79               outFile << name << '#' << sales << endl;
80
81               cout << "Salesperson's name "
82                    << "(X to stop): ";
83               getline(cin, name);
84          } //end while
85
86          outFile.close();
87      }
88      else
89          cout << "File could not be opened." << endl;
90      //end if
91  }   //end of addRecords function
92
93  void displayTotal()
94  {
95      //calculates and displays the total sales
96
97      string name = "";
98      int sales   = 0;
99      int total   = 0;
100     ifstream inFile;
101
102     //open file for input
103     inFile.open("sales.txt");
104
105     //if the open was successful, read the
106     //salesperson's name and sales amount, then add
107     //the sales amount to the accumulator, and then
108     //display the accumulator; otherwise, display
109     //an error message
```

Figure 14-20 CD collection program *(continues)*

(continued)

```
110      if (inFile.is_open())
111      {
112          getline(inFile, name, '#');
113          inFile >> sales;
114          inFile.ignore();
115
116          while (!inFile.eof())
117          {
118               total += sales;
119               getline(inFile, name, '#');
120               inFile >> sales;
121               inFile.ignore();
122          } //end while
123          inFile.close();
124          cout << "Total sales $" << total
125               << endl << endl;
126      }
127      else
128          cout << "File could not be opened." << endl;
129      //end if
130  }   //end of displayTotal function
```

Figure 14-20 CD collection program

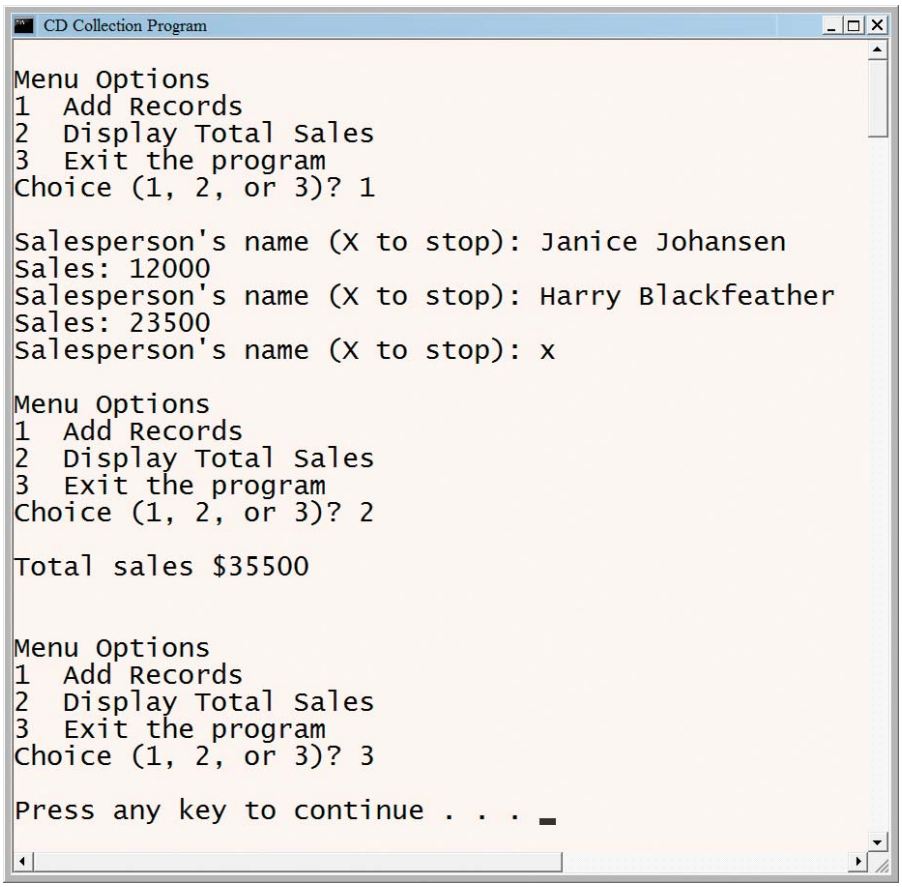

CD Collection Program
Menu Options
1 Add Records
2 Display Total Sales
3 Exit the program
Choice (1, 2, or 3)? 1
Salesperson's name (X to stop): Janice Johansen
Sales: 12000
Salesperson's name (X to stop): Harry Blackfeather
Sales: 23500
Salesperson's name (X to stop): x
Menu Options
1 Add Records
2 Display Total Sales
3 Exit the program
Choice (1, 2, or 3)? 2
Total sales $35500
Menu Options
1 Add Records
2 Display Total Sales
3 Exit the program
Choice (1, 2, or 3)? 3
Press any key to continue . . .

Figure 14-21 Sample run of the CD collection program

DIRECTIONS

Follow the instructions for starting your C++ development tool. Depending on the development tool you are using, you may need to create a new project; if so, name the project Lab14-2 Project and save it in the Cpp6\Chap14 folder. Enter the instructions shown in Figure 14-20 in a source file named Lab14-2.cpp. (Do not enter the line numbers.) Save the file in either the project folder or the Cpp6\Chap14 folder. Now, follow the appropriate instructions for running the Lab14-2.cpp file. Test the program using the data shown in Figure 14-21. If necessary, correct any bugs (errors) in the program.



LAB 14-3 Modify

If necessary, create a new project named Lab14-3 Project. Enter (or copy) the Lab14-2.cpp instructions into a new source file named Lab14-3.cpp. Change Lab14-2.cpp in the first comment to Lab14-3.cpp. Modify the menu so that it contains five options: Add Records, Display Records, Display Total Sales, Display Average Sales, and Exit the program. When the user selects the Display Records option, the program should call a function to display the contents of the sales.txt file on the screen. When the user selects the Display Average Sales option, the program should call a function to calculate and display the average sales amount stored in the file. Save and then run the program. Test the program appropriately.



LAB 14-4 Desk-Check

Desk-check the code shown in Figure 14-22, using the Lab14-4.txt file shown in Figure 14-23. What will the code in Figure 14-22 display on the screen?

```
 1  //Lab14-4.cpp - displays each region's total sales
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <fstream>
 6  using namespace std;
 7
 8  int main()
 9  {
10      int store1Sales = 0;
11      int store2Sales = 0;
12      int store1Total = 0;
13      int store2Total = 0;
14
15      //create file object and open the file
16      ifstream inFile;
17      inFile.open("Lab14-4.txt");
```

Figure 14-22 Code for Lab 14-4 *(continues)*

(continued)

```
18
19      //determine whether the file was opened
20      if (inFile.is_open())
21      {
22          //read store 1's sales amount
23          inFile >> store1Sales;
24          inFile.ignore();
25
26          while (!inFile.eof())
27          {
28               //read store 2's sales amount
29               inFile >> store2Sales;
30               inFile.ignore();
31
32               //accumulate each store's sales amount
33               store1Total += store1Sales;
34               store2Total += store2Sales;
35
36               //read store 1's sales amount
37               inFile >> store1Sales;
38               inFile.ignore();
39          } //end while
40
41          //close the file
42          inFile.close();
43
44          //display the total sales amount for each store
45          cout << "Store 1's total sales: $"
46               << store1Total << endl;
47          cout << "Store 2's total sales: $"
48               << store2Total << endl;
49      }
50      else
51          cout << "File could not be opened." << endl;
52      //end if
53
54      system("pause");
55      return 0;
56  }   //end of main function
```

your C++ development tool may not require this statement

Figure 14-22 Code for Lab 14-4

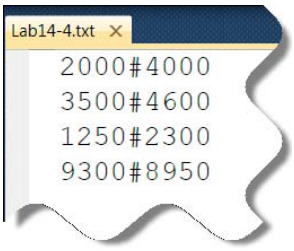


Figure 14-23 Contents of the Lab14-4.txt sequential access file



LAB 14-5 Debug

Follow the instructions for starting C++ and opening the Lab14-5.cpp file. The file is contained in either the Cpp6\Chap14\Lab14-5 Project folder or the Cpp6\Chap14 folder. Debug the program.

Summary

- Sequential access files can be either input files or output files. Input files are files whose contents are read by a program. Output files are files to which a program writes data.
- To create a file object in a program, the program must contain the `#include <fstream>` directive.
- You use the `ifstream` and `ofstream` classes, which are defined in the fstream file, to create input and output file objects, respectively. The file objects are used to represent the actual files stored on your computer's disk.
- After creating a file object, you then use the `open` function to open the file for input, output, or append.
- You can use the `is_open` function to determine whether the `open` function either succeeded or failed to open a sequential access file. The `is_open` function returns the Boolean value true if the `open` function was able to open the file. It returns the Boolean value false if the `open` function could not open the file.
- To distinguish one record from another in a sequential access file, programmers usually write each record on a separate line in the file. You do this by including the `endl` stream manipulator at the end of the statement that writes the record to the file. If the record contains more than one field, programmers use a character (such as `'#'`) to separate the data in one field from the data in another field.
- When reading data from a file, you use the `eof` function to determine whether the file pointer is at the end of the file. If the file pointer is located after the last character in the file, the `eof` function returns the Boolean value true; otherwise, it returns the Boolean value false.
- When a program is finished with a file, you should use the `close` function to close it. Failing to close an open file can result in the loss of data.



The `open`, `is_open`, and `close` functions are member functions in the `ifstream` and `ofstream` classes. The `eof` function is a member function in the `ifstream` class.

Key Terms

`!`—the Not logical operator

`close` function—closes a sequential access file in a program

`eof` function—determines whether an entire sequential access file has been read; it returns true when the file pointer is located after the last character in a sequential access file; otherwise, it returns false

Field—a single item of information about a person, place, or thing

Input files—files that contain information used as input by a program

`is_open` function—used in a program to determine whether a sequential access file was opened successfully; returns true when the open operation succeeded; otherwise, returns false

Not logical operator—reverses the truth-value of a condition

open function—used to open input and output files in a program

Output files—files that store the output produced by a program

Record—a collection of one or more related fields that contain all of the necessary data about a person, place, or thing

Sequential access files—files composed of lines of text; also referred to as text files

Text files—another name for sequential access files

Review Questions

1. A _____ is a single item of information about a person, place, or thing.
   a. field
   b. file
   c. record
   d. none of the above

2. A group of related fields that contain all of the data about a specific person, place, or thing is called a _____.
   a. field
   b. file
   c. record
   d. none of the above

3. For a program to create a file object, it must include the _____ file.
   a. fileStream
   b. fstream
   c. outFile
   d. sequential

4. You use the _____ class to instantiate an output file object.
   a. `cout`
   b. `fstream`
   c. `ofstream`
   d. `outstream`

5. Which of the following statements creates an object named `outPayroll` that represents an output file in the program?
   a. `fstream outPayroll;`
   b. `ofstream outPayroll;`
   c. `outPayroll as ofstream;`
   d. `outPayroll as outstream;`

6. Which of the following statements opens the payroll.txt file for output? The file is associated with the `outPayroll` object.
   a. `outPayroll.open("payroll.txt");`
   b. `outPayroll.open("payroll.txt", ios::out);`
   c. `outPayroll.open("payroll.txt", ios::output);`
   d. both a and b

7. To add records to the end of an existing output file, you use the _____ *mode* in the `open` function.
   a. `add`
   b. `ios::add`
   c. `ios::app`
   d. `ios::out`

8. You use the _____ function to close a sequential access file.
   a. `close`
   b. `end`
   c. `exit`
   d. `finish`

9. You can use the _____ function to determine whether the `open` function was successful.
   a. `is_open`
   b. `isopen`
   c. `isFileOpen`
   d. `is_FileOpen`

10. Which of the following statements writes the contents of the `city` variable to an output file named address.txt? The file is associated with the `outFile` object.

    a. `address.txt << city << endl;`

    b. `ofstream << city << endl;`

    c. `outFile << city << endl;`

    d. `outFile >> city >> endl;`

11. Which of the following statements reads a number from an input file named managers.txt and stores the number in the `salary` variable? The file is associated with the `inFile` object.

    a. `managers.dat << salary;`

    b. `ifstream << salary;`

    c. `inFile << salary;`

    d. none of the above

12. Which of the following statements writes the contents of the `city` and `state` variables to an output file named address.txt? The file is associated with the `outFile` object.

    a. `address.txt << city << state << endl;`

    b. `ofstream << city << state << endl;`

    c. `outFile >> city >> state >> endl;`

    d. `outFile << city << '#' << state << endl;`

13. Which of the following `while` clauses tells the computer to repeat the loop instructions until the end of the file is reached? The file is associated with the `inFile` object.

    a. `while (inFile.eof())`

    b. `while (!ifstream.eof())`

    c. `while (!inFile.eof())`

    d. `while (!ifstream.fail())`

14. Which of the following statements creates an object named `inPayroll` that represents an input file in the program?

    a. `instream inPayroll;`

    b. `ifstream inPayroll;`

    c. `inPayroll as ifstream;`

    d. `inPayroll as ifstream;`

15. Which of the following statements opens the payroll.txt file for input? The file is associated with the `inFile` object.

    a. `inFile.open("payroll.txt", ios::app);`

    b. `inFile.open("payroll.txt");`

    c. `inFile.open("payroll.txt", ios::in);`

    d. both b and c

Exercises



Pencil and Paper

TRY THIS

1. Write the statement to declare an input file object named `inSales`. (The answers to TRY THIS Exercises are located at the end of the chapter.)

TRY THIS

2. Write the statement to open a sequential access file named jansales.txt for output. The file is associated with the `outJan` object. (The answers to TRY THIS Exercises are located at the end of the chapter.)

MODIFY THIS

3. Rewrite the statement from Pencil and Paper Exercise 2 so it opens the jansales.txt file for append.

INTRODUCTORY

4. Write the statement to open a sequential access file named inventory.txt for input. The file is associated with the `inInventory` object.

INTRODUCTORY

5. Write the statement to open a sequential access file named firstQtr.txt for append. The file is associated with the `outSales` object.

INTRODUCTORY

6. Write the statement to open a sequential access file named febsales.txt for output. The file is associated with the `outFeb` object.

INTRODUCTORY

7. Write an `if` clause that determines whether an output file was opened successfully. The file is associated with the `outSales` object.

INTRODUCTORY

8. Write the statement to read a string from the sequential access file associated with the `inFile` object. Assign the string to the `textLine` variable.

INTRODUCTORY

9. Write the statement to read a number from the sequential access file associated with the `inFile` object. Assign the number to the `number` variable.

INTRODUCTORY

10. Write the statement to close the jansales.txt file, which is associated with the `outFile` object.

INTRODUCTORY

11. A program needs to write the string "Employee" and the string "Name" to the sequential access file associated with the `outFile` object. Each string should appear on a separate line in the file. Write the code to accomplish this task.

12. A program needs to write the contents of a `string` variable named `capital` and the newline character to the sequential access file associated with the `outFile` object. Write the code to accomplish this task. INTERMEDIATE

13. Write a `while` clause that tells the computer to stop processing the loop instructions when the end of the file has been reached. The file is associated with the `inFile` object. INTERMEDIATE

14. A program needs to read a sequential access file, line by line, and display each line on the computer screen. The file is associated with the `inFile` object. Write the code to accomplish this task. ADVANCED

15. Correct the condition in the following `if` clause, which should determine whether the `open` function was able to open the file associated with the `outFile` object: `if (outFile.open)`. SWAT THE BUGS



Computer

16. If necessary, create a new project named TryThis16 Project. Also create a new source file named TryThis16.cpp. Write a program that allows the user to enter the 26 letters of the alphabet. The program should save each letter on a separate line in a sequential access file named TryThis16.txt. Save and then run the program. Test the program by entering the 26 letters of the alphabet, one at a time. Verify that the program worked correctly by opening the TryThis16.txt file in a text editor. Close the TryThis16.txt file. (The answers to TRY THIS Exercises are located at the end of the chapter.) TRY THIS

17. If necessary, create a new project named TryThis17 Project. Also create a new source file named TryThis17.cpp. Write a program that saves records to a sequential access file named TryThis17.txt. Each record should contain two fields separated by the `'#'` character. The first field should contain numbers from 10 through 25. The second field should contain the square of the number in the first field. For example, the first record will contain the number 10, the `'#'` character, and the number 100. Save and then run the program. Verify that the program worked correctly by opening the TryThis17.txt file in a text editor. Close the TryThis17.txt file. (The answers to TRY THIS Exercises are located at the end of the chapter.) TRY THIS

18. In this exercise, you modify the program from TRY THIS Exercise 17. If necessary, create a new project named ModifyThis18 Project. Copy the instructions from the TryThis17.cpp file into a source file named ModifyThis18.cpp. Change the filename in the first comment to ModifyThis18.cpp. Also change the name of the sequential access file in the `open` function to ModifyThis18.txt. Modify the program so that each record contains an additional field: the cube of the number in the first field. For example, the first record should contain the number 10, the `'#'` character, the number 100, the `'#'` character, and the number 1000. Save and then run the program. Verify that the program worked correctly by opening the ModifyThis18.txt file in a text editor. Close the ModifyThis18.txt file. MODIFY THIS

INTRODUCTORY

19. If necessary, create a new project named Introductory19 Project. Also create a new source file named Introductory19.cpp. Create a program that saves a company's payroll amounts in a sequential access file. Save the amounts in fixed-point notation with two decimal places. Name the sequential access file Introductory19.txt and open the file for append. Use a negative number as the sentinel value. Save and then run the program. Enter the following payroll amounts and sentinel value: 45678.99, 67000.56, and -1. Now, run the program again. This time, enter the following payroll amounts and sentinel value: 25000.89, 35600.55, and -1. Open the Introductory19.txt file in a text editor. The file should contain four payroll amounts, with each amount appearing on a separate line in the file. Close the Introductory19.txt file.

INTRODUCTORY

20. If necessary, create a new project named Introductory20 Project. Also create a new source file named Introductory20.cpp. Create a program that saves prices in a sequential access file. Save the prices in fixed-point notation with two decimal places. Name the sequential access file Introductory20.txt and open the file for append. Use a negative number as the sentinel value. Save and then run the program. Enter the following prices and sentinel value: 10.50, 15.99, and -1. Now, run the program again. This time, enter the following prices and sentinel value: 20, 76.54, 17.34, and -1. Open the Introductory20.txt file in a text editor. The file should contain five prices, with each price appearing on a separate line in the file. Close the Introductory20.txt file.

INTRODUCTORY

21. If necessary, create a new project named Introductory21 Project. Also create a new source file named Introductory21.cpp. If you are using Microsoft Visual C++, copy the Introductory21.txt file from the Cpp6\Chap14 folder to the Cpp6\Chap14\Introductory21 Project folder. Use a text editor to open the Introductory21.txt file, which contains letters of the alphabet. Close the Introductory21.txt file. Create a program that counts the number of letters stored in the file. The program should display the number of letters on the computer screen. Save and then run the program.

INTERMEDIATE

22. If necessary, create a new project named Intermediate22 Project. Also create a new source file named Intermediate22.cpp. If you are using Microsoft Visual C++, copy the Intermediate22.txt file from the Cpp6\Chap14 folder to the Cpp6\Chap14\Intermediate22 Project folder. Use a text editor to open the Intermediate22.txt file, which contains payroll amounts. Close the Intermediate22.txt file. Create a program that calculates and displays the total of the payroll amounts stored in the file. Display the total with a dollar sign and two decimal places. Save and then run the program.

INTERMEDIATE

23. If necessary, create a new project named Intermediate23 Project. Also create a new source file named Intermediate23.cpp. If you are using Microsoft Visual C++, copy the Intermediate23.txt file from the Cpp6\Chap14 folder to the Cpp6\Chap14\Intermediate23 Project folder. Use a text editor to open the Intermediate23.txt file, which contains prices. Close the Intermediate23.txt file. Create a program

that calculates and displays the average price stored in the file. Display the average with a dollar sign and two decimal places. Save and then run the program.

24. If necessary, create a new project named Intermediate24 Project. Also create a new source file named Intermediate24.cpp. If you are using Microsoft Visual C++, copy the Intermediate24.txt file from the Cpp6\Chap14 folder to the Cpp6\Chap14\Intermediate24 Project folder. Use a text editor to open the Intermediate24.txt file, which contains payroll codes and salaries. Close the Intermediate24.txt file. Create a program that allows the user to enter a payroll code. The program should search for the payroll code in the file and then display the appropriate salary. If the payroll code is not in the file, the program should display an appropriate message. Use a sentinel value to end the program. Save and then run the program. Test the program by entering the following payroll codes: 10, 24, 55, 32, and 6. Stop the program by entering your sentinel value. INTERMEDIATE

25. If necessary, create a new project named Advanced25 Project. Also create a new source file named Advanced25.cpp. If you are using Microsoft Visual C++, copy the Advanced25.txt file from the Cpp6\Chap14 folder to the Cpp6\Chap14\Advanced25 Project folder. Use a text editor to open the Advanced25.txt file, which contains the names of the items in inventory, as well as each item's quantity and price. Close the Advanced25.txt file. Write a program that displays the contents of the file in three columns titled "Name", "Quantity", and "Price". The program also should display a fourth column that contains the result of multiplying each item's quantity by its price. Use "Value" as the column's title. (Hint: Use `'\t'`, which is the escape sequence for the tab key, to align the columns.) In addition, the program should calculate and display the total value of the items in inventory. Display the price, value, and total value with two decimal places. Save and then run the program. ADVANCED

26. If necessary, create a new project named Advanced26 Project. Also create a new source file named Advanced26.cpp. Write a program that allows the user to record the names of cities and their corresponding ZIP codes in a sequential access file named Advanced26.txt. The program also should allow the user to look up a ZIP code in the file and display the name of its corresponding city. If the ZIP code is not in the file, the program should display an appropriate message. Save and then run the program. Enter the ZIP codes and city names listed in Figure 14-24. Next, display the name of the city corresponding to the following ZIP codes: 60135, 60544, and 55555. ADVANCED

ZIP code	City
60561	Darien
60544	Hinsdale
60137	Glen Ellyn
60135	Downers Grove
60136	Burr Ridge

Figure 14-24

ADVANCED

27. If necessary, create a new project named Advanced27 Project. Also create a new source file named Advanced27.cpp. If you are using Microsoft Visual C++, copy the Advanced27.txt file from the Cpp6\Chap14 folder to the Cpp6\Chap14\Advanced27 Project folder. Each salesperson at BobCat Motors is assigned a code that consists of two characters. The first character is either the letter F (which indicates a full-time employee) or the letter P (which indicates a part-time employee). The second character is either a 1 (indicating the salesperson sells new cars) or a 2 (indicating the salesperson sells used cars). Use a text editor to open the Advanced27.txt file, which contains the names of BobCat's salespeople along with each salesperson's code, then close the file. Write a program that prompts the user to enter the code (F1, F2, P1, or P2). The program should search the Advanced27.txt file for the code and then display only the names of the salespeople assigned that code. Display an appropriate message if the user enters an invalid code. Save and then run the program. Test the program by entering F2 as the code. The program should display three records: Mary Jones, Joel Adkari, and Janice Paulo. Now, test the program using codes of F1, P1, P2, and S3.

ADVANCED

28. If necessary, create a new project named Advanced28 Project. Also create a new source file named Advanced28.cpp. If you are using Microsoft Visual C++, copy the Advanced28.txt file from the Cpp6\Chap14 folder to the Cpp6\Chap14\Advanced28 Project folder. Use a text editor to open the Advanced28.txt file, which contains 20 numbers. Close the Advanced28.txt file. Write a program that performs the following for each number in the Advanced28.txt file: read the number, add 1 to the number, write the new number to another sequential access file named UpdatedAdvanced28.txt. Save and then run the program. Use a text editor to open the UpdatedAdvanced28.txt file. Each number in the file should be one greater than its corresponding number in the Advanced28.txt file. Close the UpdatedAdvanced28.txt file.

ADVANCED

29. If necessary, create a new project named Advanced29 Project. Also create a new source file named Advanced29.cpp. If you are using Microsoft Visual C++, copy the Advanced29.txt file from the Cpp6\Chap14 folder to the Cpp6\Chap14\Advanced29 Project folder. Use a text editor to open the Advanced29.txt file, which contains 12 numbers. Close the Advanced29.txt file. Write a program that reads the numbers contained in the Advanced29.txt file and writes only the even numbers to a new sequential access file named EvenAdvanced29.txt. Save and then run the program. Use a text editor to open the EvenAdvanced29.txt file, which should contain only the even numbers. Close the EvenAdvanced29.txt file.

SWAT THE BUGS

30. Follow the instructions for starting C++ and opening the SwatTheBugs30.cpp file. The file is contained in either the Cpp6\Chap14\SwatTheBugs30 Project folder or the Cpp6\Chap14 folder. Debug the program.

Answers to TRY THIS Exercises

Pencil and Paper

1. `ifstream inSales;`

2. `outJan.open("jansales.txt");` or `outJan.open("jansales.txt", ios::out);`

Computer

16. See Figures 14-25 and 14-26.

```
1  //TryThis16.cpp - writes 26 letters to a sequential
2  //access file
3  //Created/revised by <your name> on <current date>
4
5  #include <iostream>
6  #include <fstream>
7  using namespace std;
8
9  int main()
10 {
11     char letter = ' ';
12
13     //create file object and open the file
14     ofstream outFile;
15     outFile.open("TryThis16.txt");
16
17     //determine whether the file was opened
18     if (outFile.is_open())
19     {
20         //get the 26 letters of the alphabet and
21         //write each to the file
22         for (int x = 1; x <= 26; x = x + 1)
23         {
24              cout << "Enter letter " << x << ": ";
25              cin >> letter;
26              outFile << letter << endl;
27         }    //end for
28
29         //close the file
30         outFile.close();
31     }
32     else
33         cout << "The file could not be opened." << endl;
34     //end if
35
36     system("pause");
37     return 0;
38 }   //end of main function
```

your C++ development tool may not require this statement



In Line 22, you also can use either `x += 1` or `x++`.

Figure 14-25

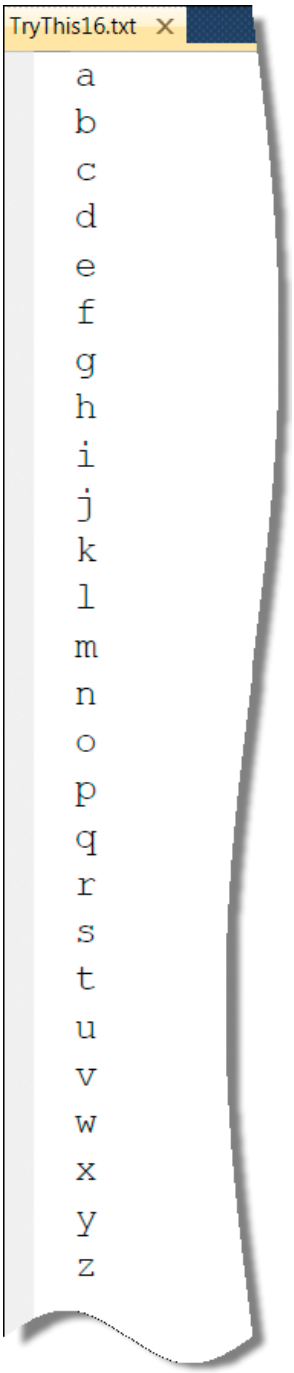


Figure 14-26

17. See Figures 14-27 and 14-28.

```
1  //TryThis17.cpp - saves numbers from 10 through
2  //25, along with the square of each number,
3  //in a sequential access file
4  //Created/revised by <your name> on <current date>
5
6  #include <iostream>
```

Figure 14-27 *(continues)*

(continued)

```
 7  #include <cmath>
 8  #include <fstream>
 9  using namespace std;
10
11 int main()
12 {
13      ofstream outNumbers;
14      outNumbers.open("TryThis17.txt");
15
16      if (outNumbers.is_open())
17      {
18          for (int x = 10; x < 26; x += 1)
19               outNumbers << x << '#'
20                    << pow(x, 2.0) << endl;
21          //end for
22          outNumbers.close();
23      }
24      else
25          cout << "The file could not be opened."
26               << endl;
27      //end if
28
29      system("pause");
30      return 0;
31  }   //end of main function
```

your C++ development tool may not require this statement



In Line 18, you also can use either `x++` or `x = x + 1`.

Figure 14-27

TryThis17.txt

```
10#100
11#121
12#144
13#169
14#196
15#225
16#256
17#289
18#324
19#361
20#400
21#441
22#484
23#529
24#576
25#625
```

Figure 14-28

APPENDIX A

Answers to Mini-Quizzes and Labs

Answers to Chapter 1 Mini-Quizzes

Mini-Quiz 1-1

1. machine
2. a. a procedure-oriented
3. b. an object-oriented
4. compiler

Mini-Quiz 1-2

1. sequence, selection, repetition
2. sequence
3. algorithm
4. repetition
5. repetition
6. selection

Answers to Chapter 1 Labs



LAB 1-1 Stop and Analyze

1. sequence and repetition
2. Mary Smith and 60
3. change the last instruction to `print the salesperson's name, sales amount, and bonus amount`

4. The modifications are shaded in the algorithm.

```
repeat for each salesperson
        enter the salesperson's name and sales amount
        calculate the bonus amount by multiplying the sales
         amount by 3%
        print the salesperson's name and bonus amount
end repeat
```

5. The modifications are shaded in the algorithm.

```
enter the bonus rate
repeat 5 times
        enter the salesperson's name and sales amount
        calculate the bonus amount by multiplying the sales amount
        by the bonus rate
        print the salesperson's name and bonus amount
end repeat
```



LAB 1-2 Plan and Create

```
repeat 25 times
        read the student's answer and the correct answer
        if (the student's answer is not the same as the correct answer)
                mark the student's answer incorrect
        end if
end repeat
```



LAB 1-3 Modify

You can use either of the following algorithms. The modifications are shaded in each.

Algorithm 1

```
repeat 5 times
        enter the salesperson's name and sales amount
        if (the sales amount is greater than 2000)
                calculate the bonus amount by multiplying the sales
                amount by 3.5%
```

```
        else
            calculate the bonus amount by multiplying the sales amount
            by 3%
        end if
        print the salesperson's name and bonus amount
end repeat
```

Algorithm 2

```
repeat 5 times
        enter the salesperson's name and sales amount
        if (the sales amount is less than or equal to 2000)
            calculate the bonus amount by multiplying the sales amount
            by 3%
        else
            calculate the bonus amount by multiplying the sales amount
            by 3.5%
        end if
        print the salesperson's name and bonus amount
end repeat
```

Answers to Chapter 2 Mini-Quizzes

Mini-Quiz 2-1

1.	Output:	sales tax
	Input:	purchase amount sales tax rate
	Missing information:	none
2.	Output:	savings
	Input:	number of CDs purchased club CD price store CD price
	Missing information:	store CD price
3.	Output:	total amount saved in January
	Input:	amount saved per day number of days in January
	Missing information:	none (Although the number of days in January is not specified in the problem specification, that information can be found in any calendar.)
4.	Output:	yearly savings
	Input:	amount saved per day number of days in the year
	Missing information:	number of days in the year (Because some years are leap years, you would need to know the number of days in the year.)

Mini-Quiz 2-2

1. input/output

2. rectangular

3.

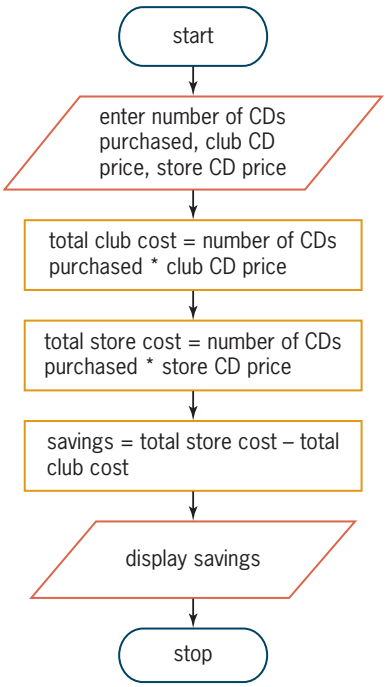
Input	Processing	Output
purchase amount sales tax rate	Processing items: none	sales tax

Algorithm:
1. enter the purchase amount and sales tax rate
2. calculate the sales tax by multiplying the purchase amount by the sales tax rate
3. display the sales tax

4.

Input	Processing	Output
number of CDs purchased club CD price store CD price	Processing items: total club cost total store cost	savings

Algorithm:


start
enter number of CDs purchased, club CD price, store CD price
total club cost = number of CDs purchased * club CD price
total store cost = number of CDs purchased * store CD price
savings = total store cost – total club cost
display savings
stop

Mini-Quiz 2-3

1.

purchase amount	sales tax rate	sales tax
~~67~~	~~.05~~	~~3.35~~
100	.02	2.00

2.

first number	second number	sum	average
~~5~~	~~11~~	~~16~~	~~8~~
6	12	18	9

Answers to Chapter 2 Labs



LAB 2-1 Stop and Analyze

1. The algorithm will display 306 when the user enters 300 and .02 as the current weekly pay and raise percentage, respectively. The algorithm will display 512.50 when the user enters 500 and .025 as the current weekly pay and raise percentage, respectively.

2.

Input	Processing	Output
current weekly pay raise percentage	Processing items: raise	new weekly pay

Algorithm:
1. enter the current weekly pay and raise percentage
2. calculate the raise by multiplying the current weekly pay by the raise percentage
3. calculate the new weekly pay by adding the raise to the current weekly pay
4. display the new weekly pay

current weekly pay	raise percentage	raise	new weekly pay
~~300~~	~~.02~~	~~6~~	~~306~~
500	.025	12.50	512.50

3.

Input	Processing	Output
current weekly pay raise percentage	Processing items: none	raise new weekly pay

Algorithm:
1. enter the current weekly pay and raise percentage
2. calculate the raise by multiplying the current weekly pay by the raise percentage
3. calculate the new weekly pay by adding the raise to the current weekly pay
4. display the raise and new weekly pay

current weekly pay	raise percentage	raise	new weekly pay
~~300~~	~~.02~~	~~6~~	~~306~~
500	.025	12.50	512.50



LAB 2-2 Plan and Create

No answer required.



LAB 2-3 Modify

Input
number of nights
per-night rate
room service charge
telephone charge
entertainment tax rate

Processing
Processing items:
room charge
entertainment tax

Output
total bill

Algorithm:
1. enter the number of nights, per-night rate, room service charge, telephone charge, and entertainment tax rate
2. calculate the room charge by multiplying the number of nights by the per-night rate
3. calculate the entertainment tax by multiplying the room charge by the entertainment tax rate
4. calculate the total bill by adding together the room charge, entertainment tax, room service charge, and telephone charge
5. display the total bill

number of nights	per-night rate	room service charge	telephone charge	entertainment tax rate
~~3~~	~~70~~	~~0~~	~~10~~	~~.05~~
7	100	25	6	.03

room charge	entertainment tax	total bill
~~210~~	~~10.50~~	~~230.50~~
700	21	752



LAB 2-4 Desk-Check

assessed value	tax rate	annual property tax
~~104000~~	~~1.50~~	~~1560~~
~~239000~~	~~1.15~~	~~2748.50~~
86000	.98	842.80



LAB 2-5 Debug

first number	second number	third number	sum	average
~~25~~	~~63~~	~~14~~	~~102~~	~~34~~
33	56	70	159	53

Answers to Chapter 3 Mini-Quizzes

Mini-Quiz 3-1

1. one
2. a. `quantity`
3. c. `COMMISSION_RATE`
4. variables and named constants

Mini-Quiz 3-2

1. b. False
2. c. 100000

3. 98, 01100010
4. d. both a and c

Mini-Quiz 3-3

1. b. '%'
2. d. all of the above
3. 10
4. `int population = 0;`
5. `const double MAX_PAY = 25.55;`

Answers to Chapter 3 Labs



LAB 3-1 Stop and Analyze

1. The problem requires three memory locations.
2. The problem requires three variables, but no named constants. Variables were chosen so that the values of the input and output items can vary during runtime.
3. The input and output items could be stored in either `float` or `double` memory locations.
4.
```
double currentPay = 0.0;
double raiseRate = 0.0;
double newPay = 0.0;
```
5. `const double RAISE_RATE = .02;`



LAB 3-2 Plan and Create

No answer required.



LAB 3-3 Modify

The modifications made to Figure 3-17 are shaded in the IPO chart.

Input	Processing	Output
radius pi (3.14)	Processing items: radius squared Algorithm: 1. enter the radius 2. calculate the radius squared by multiplying the radius by itself 3. calculate the area by multiplying the radius squared by pi 4. display the area	area

The modifications made to Figure 3-20 are shaded in the IPO chart information and C++ instructions.

IPO chart information	C++ instructions
Input	
radius	`double radius = 0.0;`
pi (3.14)	`const double PI = 3.14;`
Processing	
radius squared	`double radiusSquared = 0.0;`
Output	
area	`double area = 0.0;`



LAB 3-4 Desk-Check

The modifications made to Figure 3-18 are shaded in the manual calculations.

First desk-check

```
   4   (radius)
*  4   (radius)
  16   (radius squared)
*  3.14  (pi)
  50.24  (area)
```

Second desk-check

```
   5.5   (radius)
*  5.5   (radius)
  30.25  (radius squared)
*  3.14  (pi)
  94.985 (area)
```

The modifications made to Figure 3-19 are shaded in the desk-check table.

radius	pi	radius squared	area
~~4~~	~~3.14~~	~~16~~	~~50.24~~
5.5	3.14	30.25	94.985



LAB 3-5 Debug

The modifications made to Figure 3-21 are shaded in the C++ instructions column.

IPO chart information	C++ instructions
Input	
first number	`double first = 0.0;`
second number	`double second = 0.0;`
third number	`double third = 0.0;`
Processing	
sum	`double sum = 0.0;`
Output	
average	`double average = 0.0;`

Answers to Chapter 4 Mini-Quizzes

Mini-Quiz 4-1

1. b. `cin >> grossPay;`
2. c. `cout << grossPay;`
3. d. all of the above
4. `<<`

Mini-Quiz 4-2

1. `grossPay = 9.55 * hours;`
2. `grossPay = 9.55 * static_cast<double>(hours);`
3. The expression evaluates to 13.5. It should evaluate to 15.75. The expression evaluates incorrectly because dividing the integer 7 by the integer 2 results in the integer 3 rather than in the `double` number 3.5. Multiplying the integer 3 by the `double` number 4.5 results in the incorrect answer of 13.5.
4. You can use any of the following expressions. You also can use the `static_cast` operator to type cast at least one of the integers in the expression.

```
7.0 / 2.0 * 4.5
7.0 / 2 * 4.5
7 / 2.0 * 4.5
```


Mini-Quiz 4-3

1. syntax
2. c. source
3. 10
4. `age += 1;`

Answers to Chapter 4 Labs



LAB 4-1 Stop and Analyze

1. When evaluating the `totalCost = numberOfPeople * costPerPerson;` statement, the computer converts the integer stored in the `numberOfPeople` variable to the `double` number 10.0 before multiplying it by the value stored in the `costPerPerson` variable (the `double` number 7.45). The result is the `double` number 74.5, which the computer assigns to the `double` variable `totalCost`. The value assigned to the `totalCost` variable is correct.
2. When evaluating the `numberOfPeople = numberOfPeople / 2;` statement, the computer divides the integer 10 by the integer 2. The result is the integer 5, which the computer assigns to the `numberOfPeople` variable. The value assigned to the `numberOfPeople` variable is correct.
3. When evaluating the `costPerPerson = costPerPerson + 3;` statement, the computer converts the integer 3 to the `double` number 3.0 before adding it to the value stored in the `costPerPerson` variable (the `double` number 7.45). The result is the `double` number 10.45, which the computer assigns to the `costPerPerson` variable. The value assigned to the `costPerPerson` variable is correct.
4. When evaluating the `average = score1 + score2 / 2;` statement, the computer converts the integer 2 to the `double` number 2.0 before dividing it into the value stored in the `score2` variable (the `double` number 90.0). The result is the `double` number 45.0, which the computer adds to the value stored in the `score1` variable (the `double` number 100.0). The result is the `double` number 145.0, which the computer assigns to the `double` variable `average`. The

value assigned to the `average` variable is incorrect. You can fix the statement as follows: `average = (score1 + score2) / 2;`

5. When evaluating the `avgSales = (juneSales + julySales) / 2;` statement, the computer adds the integer stored in the `juneSales` variable (933) to the integer stored in the `julySales` variable (1216). The result is the integer 2149, which the computer divides by the integer 2. The quotient is the integer 1074, which the computer converts to the `double` number 1074.0 before assigning it to the `double` variable `avgSales`. The value assigned to the `avgSales` variable is incorrect. Two ways you can fix the statement are shown here:

```
avgSales = (juneSales + julySales) / 2.0;
avgSales = static_cast<double>(juneSales
+ julySales) / 2;
```



LAB 4-2 Plan and Create

No answer required.



LAB 4-3 Modify

The modifications made to the Lab4-2.cpp file are shaded.

```
//Lab4-3.cpp - displays the total owed
//Created/revised by <your name> on <current date>

#include <iostream>
using namespace std;

int main()
{
     //declare variables and named constants
     double hours           = 0.0;
     double totalOwed       = 0.0;
     const int FEE_PER_HOUR = 105;
     const int ROOM_BOARD   = 2000;

     //enter hours enrolled
     cout << "Hours enrolled? ";
     cin >> hours;

     //calculate total owed
     totalOwed = hours * FEE_PER_HOUR + ROOM_BOARD;

     //display total owed
     cout << "Total owed: $" << totalOwed << endl;
```

```
    system("pause");
    return 0;
}   //end of main function
```




LAB 4-4 Desk-Check

num	answer
75	0
	1

LAB 4-5 Debug

To debug the program, the student needs to convert at least one of the items on the right side of the assignment operator in the `raise = salary * 3 / 100;` statement to the `double` data type. For example, the student can use `raise = salary * 3 / 100.0;`, `raise = salary * 3.0 / 100`, or `static_cast<double>(salary) * 3 / 100;`, or `static_cast<double>(salary) * 3.0 / 100.0;`.

Answers to Chapter 5 Mini-Quizzes

Mini-Quiz 5-1

1. end if
2. b. False
3. a. diamond

Mini-Quiz 5-2

1. d. none of the above
2. c. `if (quantity == 100)`
3. a. `if (sales >= 300.99)`
4. c. `!=`

Mini-Quiz 5-3

1. true
2. false
3. false
4. b. `if (age >= 30 && age <= 40)`
5. a. `if (code == 'R' || code == 'r')`

Mini-Quiz 5-4

1. d. `cout << fixed << setprecision(2);`
2. c. `letter = tolower(letter);`
3. d. 34.650000

Answers to Chapter 5 Labs



LAB 5-1 Stop and Analyze

1. The number .03 will be assigned to the `rate` variable when the user enters a pay grade of 1. The number .02 will be assigned to the `rate` variable when the user enters either 3 or 5 as the pay grade.
2. The directive on Line 5 is necessary because the program uses the `setprecision` stream manipulator.
3. The 1 in the `if` statement is enclosed in single quotation marks because the `payGrade` variable's data type is `char`.
4.
```
if (payGrade != '1')
     rate = .02;
else
     rate = .03;
//end if
```
5. You also can write the `salary = salary + salary * rate;` statement as follows:

```
salary = salary * (1 + rate);
```

6. You would need to remove the `double rate = 0.0;` and `salary = salary + salary * rate;` statements. You also would need to change the `rate = .03;` statement to `salary = salary + salary * .03;`, and change the `rate = .02;` statement to `salary = salary + salary * .02;`.



LAB 5-2 Plan and Create

No answer required.



LAB 5-3 Modify

```
if (totalCals < 0 || fatGrams < 0)
     cout << "Input error" << endl;
else
{
     //calculate and display the output
     fatCals = fatGrams * 9;
     fatPercent = static_cast<double>(fatCals)
          / static_cast<double>(totalCals) * 100;
     cout << "Fat calories: " << fatCals << endl;
     cout << fixed << setprecision(0);
     cout << "Fat percentage: " << fatPercent
     << "%" << endl;
}    //end if
```



LAB 5-4 Desk-Check

```
letter
P
```

When the user enters the letter P, the compound condition in the first `if` statement evaluates to true, and the statement's true path displays the "Pass" message on the computer screen. Although the correct message already appears on the screen, the computer still evaluates the second `if` statement's compound condition, which determines whether to display the "Fail" message. The second evaluation is unnecessary and makes the code inefficient. You can fix the code by deleting the first `//end if` comment and replacing the `if (letter != 'P' || letter != 'p')` clause with `else`.



LAB 5-5 Debug

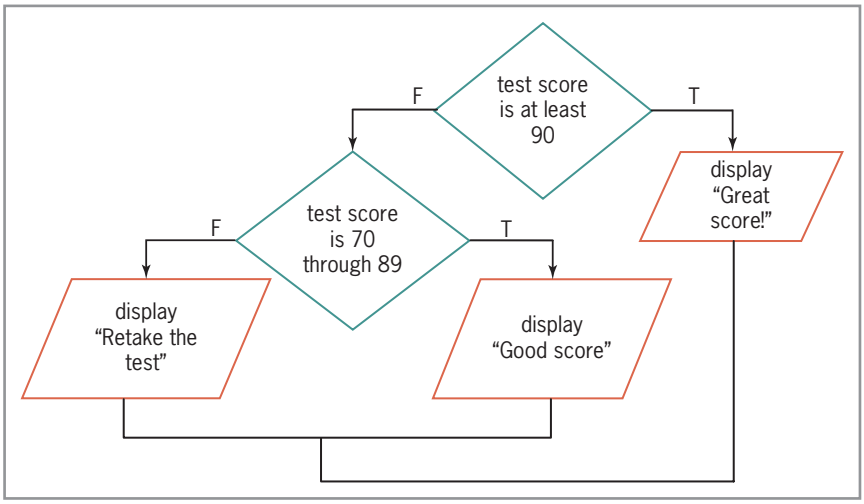
To debug the program, the student needs to change `if (code = '2')` to `if (code == '2')`.

Answers to Chapter 6 Mini-Quizzes

Mini-Quiz 6-1

1.
```
if (the test score is at least 90)
   display "Great score!"
else
   if (the test score is 70 through 89)
      display "Good score"
   else
      display "Retake the test"
   end if
end if
```

2.



3.
```
if (score >= 90)
    cout << "Great score!" << endl;
 else
    if (score >= 70)
       cout << "Good score" << endl;
    else
       cout << "Retake the test" << endl;
    //end if
 //end if
```

4. a. membership status, day of the week

Mini-Quiz 6-2

1. Using a compound condition rather than a nested selection structure
   Reversing the decisions in the outer and nested selection structures
   Using an unnecessary nested selection structure
2. Using an unnecessary nested selection structure

Mini-Quiz 6-3

1.
```
if (score >= 90)
   cout << "Great score!" << endl;
else if (score >= 70)
   cout << "Good score" << endl;
else if (score >= 0)
   cout << "Retake the test" << endl;
else
   cout << "Invalid test score" << endl;
//end if
```

2. b. `case 2:`

3. `break`

Answers to Chapter 6 Labs



LAB 6-1 Stop and Analyze

6. The program will display the number 35 when the code is C.

7.
```
if (code == 'S')
   fee = 40;
else
   if (code == 'F')
      fee = 50;
   else
      if (code == 'A')
         fee = 30;
      else
         if (code == 'C')
            fee = 35;
         else
            fee = 0;
         //end if
      //end if
   //end if
//end if
```

8.
```
if (code == 'S')
   fee = 40;
else if (code == 'F')
   fee = 50;
else if (code == 'A')
   fee = 30;
else if (code == 'C')
   fee = 35;
else
   fee = 0;
//end if
```

9.
```
switch (code)
{
case 'S':
    fee = 40;
    break;
case 'F':
    fee = 50;
    break;
case 'A':
    fee = 30;
    break;
case 'C':
    fee = 35;
    break;
default;
    fee = 0;
}   //end switch
```

10.
```
switch (code)
{
case 'S':
    cout << 40 << endl;
    break;
case 'F':
    cout << 50 << endl;
    break;
case 'A':
    cout << 30 << endl;
    break;
case 'C':
    cout << 35 << endl;
    break;
default;
    cout << "Invalid code";
}   //end switch
```



LAB 6-2 Plan and Create

No answer required.



LAB 6-3 Modify

```
//Lab6-3.cpp - displays a salesperson's commission
//Created/revised by <your name> on <current date>

#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   //declare variables
   int sales      = 0;
   double commission = 0.0;
   char code      = ' ';

   //enter the code
   cout << "Code (1, 2, or 3): ";
   cin >> code;

   //determine whether the code is valid
   if (code != '1' && code != '2' && code != '3')
      cout << "Invalid code" << endl;
   else
   {
      //enter the sales
      cout << "Sales: ";
      cin >> sales;

      //determine whether the sales are valid
      if (sales < 0)
         cout << "The sales cannot be less than 0." << endl;
      else
      {
         //calculate and display the commission
         switch (code)
         {
         case '1':
            commission = sales * .02;
            break;
         case '2':
            commission = (sales - 100000) * .05 + 2000;
            break;
         case '3':
            commission = (sales - 400000) * .1 + 17000;
            break;
         }  //end switch
```

```
         cout << fixed << setprecision(2);
         cout << "Commission: $" << commission << endl;
      }  //end if
   }  //end if

   system("pause");
   return 0;
}  //end of main function
```



LAB 6-4 Desk-Check

number
~~0~~
~~2~~
~~4~~
~~0~~
~~5~~
~~10~~
~~0~~
~~100~~
50



LAB 6-5 Debug

The modifications are shaded in the code.

```
//Lab6-5.cpp - displays the salary associated with a code
//Codes        Salary
//1            $45,000
//2, 5         $33,000
//3, 4         $25,000
//Created/revised by <your name> on <current date>

#include <iostream>
using namespace std;

int main()
{
   //declare variable
   int code = 0;

   //get code
   cout << "Enter the code (1 through 5): ";
   cin >> code;

   //display salary
   if (code == 1)
      cout << "$45,000" << endl;
```

```
   else if (code == 2 || code == 5)
      cout << "$33,000" << endl;
   else if (code == 3 || code == 4)
      cout << "$25,000" << endl;
   else
      cout << "Entry error" << endl;
   //end if

   system("pause");
   return 0;
}  //end of main function
```

Answers to Chapter 7 Mini-Quizzes

Mini-Quiz 7-1

1. `while (quantity > 0)`

2. `while (quantity >= 0)`

3. `while (inStock > reorder)`

4. `while (toupper(letter) == 'Y')` [You also can use `while (tolower(letter) == 'y')` or `while (letter == 'Y' || letter == 'y')`.

5. a. -9

Mini-Quiz 7-2

1. a. accumulators

2. `quantity += 2;` (or `quantity = quantity + 2;`)

3. `total -= 3;` (or `total = total - 3;` or `total += -3;` or `total = total + -3;`)

4. `totalPurchases += purchases;` (or `totalPurchases = totalPurchases + purchases;`)

Mini-Quiz 7-3

1.
```
while (evenNum < 9)
{
        cout << evenNum << endl;
        evenNum += 2;      (or evenNum = evenNum + 2)
}       //end while
```

2. a. `for (int x = 10; x <= 100; x = x + 10)`

3. d. 110

4. `for (int x = 25; x > 0; x = x - 5)` (You also can use `x -= 5` as the *update* argument.)

5. 0

6. `for (int num = 2; num < 9; num += 2)` (You also can use `num = num + 2` as the *update* argument.)

Answers to Chapter 7 Labs



LAB 7-1 Stop and Analyze

1. The program inputs a temperature. It uses two processing items: a counter that keeps track of the number of temperatures and an accumulator that totals the temperatures. The program displays the average temperature.
2. The number 999 was chosen as a sentinel value because it's not a valid outside temperature. A negative number was not chosen as a sentinel value because the temperature outside can be below zero.
3. The selection structure on Lines 32 through 43 is necessary to prevent the program from calculating the average temperature when the counter variable (`numberOfTemps`) contains the number 0. Dividing by zero causes the program to end abruptly with an error.
4. Although the statement on Lines 35 and 36 doesn't need both type casts, it does need at least one of them. Without at least one type cast, the quotient obtained when dividing the contents of the `totalTemp` variable by the contents of the `numberOfTemps` variable would be an integer (rather than a `double` number). This is because both of those variables have the `int` data type; when you divide two integers, the result is an integer. At least one type cast is needed to force the computer to perform the division using `double` numbers, which will result in a quotient that also is a `double` number.
5. The `cout << fixed << setprecision(1);` statement on Line 37 tells the computer to display the output in fixed-point notation with one decimal place.

6. The `numberOfTemps` counter variable keeps track of the number of temperatures entered. Therefore, it should be initialized to 0 at the beginning of the program and then updated only when a temperature has been entered.

7. The average temperature is 76.666... (with the 6 repeating).

numberOfTemps	totalTemp	temp	average
~~0~~	~~0~~	~~0~~	~~0.0~~
~~1~~	~~78~~	~~78~~	
~~2~~	~~163~~	~~85~~	
3	230	~~67~~	
		999	76.666...

8. The program displays 76.7.

9. The program displays the "No temperatures were entered." message.

10. The program displays the "No temperatures were entered." message.

```
//Lab7-1.cpp - calculates and displays the average temperature
//Created/revised by <your name> on <current date>

#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   //declare variables
   int numberOfTemps = 0; //counter
   int totalTemp     = 0; //accumulator
   int temp          = 0;
   double average    = 0.0;

   //get first temperature
   cout << "First temperature (999 to stop): ";
   cin >> temp;

   for (; temp != 999;)
   {
      //update counter and accumulator
      numberOfTemps += 1;
      totalTemp += temp;
      //get remaining temperatures
      cout << "Next temperature (999 to stop): ";
      cin >> temp;
   }  //end for

   //verify that at least one temperature was entered
   if (numberOfTemps > 0)
   {
      //calculate and display average temperature
      average = static_cast<double>(totalTemp) /
         static_cast<double>(numberOfTemps);
      cout << fixed << setprecision(1);
      cout << endl << "Average temperature: "
         << average << endl;
   }
```

```
   else
      cout << "No temperatures were entered." << endl;
   //end if

   system("pause");
   return 0;
}  //end of main function
```

11. The average temperature is 14.8.



LAB 7-2 Plan and Create

No answer required.



LAB 7-3 Modify

The program displays the correct total points (255) and grade (C). The program shows that the professor entered 7 scores.

```
//Lab7-3.cpp - displays the total points earned and grade
//Created/revised by <your name> on <current date>

#include <iostream>

using namespace std;

int main()
{
   //declare variables
   int score       = 0;
   int totalPoints = 0; //accumulator
   char grade      = ' ';
   int numScores   = 0; //counter

   //get first score
   cout << "First score (-1 to stop): ";
   cin >> score;

   while (score != -1)
   {
      //update accumulator and counter, then get another score
      totalPoints += score;
      numScores += 1;
      cout << "Next score (-1 to stop): ";
      cin >> score;
   }  //end while

   //determine grade
   if (totalPoints >= 315)
      grade = 'A';
```

```
   else if (totalPoints >= 280)
      grade = 'B';
   else if (totalPoints >= 245)
      grade = 'C';
   else if (totalPoints >= 210)
      grade = 'D';
   else
      grade = 'F';
   //end if

   //display the total points, grade, and number of scores
   cout << "Total points earned: " << totalPoints << endl;
   cout << "Grade: " << grade << endl;
   cout << "Number of scores: " << numScores << endl;
   system("pause");
   return 0;
}  //end of main function
```



LAB 7-4 Desk-Check

First desk-check:

squaredNumber	number
~~0~~	~~1~~
~~1~~	~~2~~
~~4~~	~~3~~
~~9~~	~~4~~
16	5

Corrected code:

```
//declare variables
int squaredNumber = 0;

for (int number = 1; number <= 5; number = number + 1)
{
      squaredNumber = number * number;
      cout << squaredNumber << endl;
}     //end for
```

Second desk-check:

squaredNumber	number
~~0~~	~~1~~
~~1~~	~~2~~
~~4~~	~~3~~
~~9~~	~~4~~
~~16~~	~~5~~
25	6



LAB 7-5 Debug

To debug the program, enter a `cin >> price;` statement below the `cout << "Next price: ";` statement.

Answers to Chapter 8 Mini-Quizzes

Mini-Quiz 8-1

1. 4
2. 1
3. 3
4. 1

Mini-Quiz 8-2

1. b. False
2. d. semicolon
3. `} while (inStock > reorder);`
4. `} while (toupper(letter) == 'Y');`

Mini-Quiz 8-3

1. b. False
2. a. True
3. a. outer, nested
4. d. all of the above

Answers to Chapter 8 Labs



LAB 8-1 Stop and Analyze

1. The program contains two pretest loops. It does not contain any posttest loops.
2. The loop that keeps track of the region number is controlled by a counter. The loop's condition will evaluate to true for the following counter values: 1 and 2. A counter value of 3 will make the loop's condition evaluate to false.
3. The loop that keeps track of the sales amounts is controlled by a sentinel value. The valid sentinel values for the loop are any numbers that are less than or equal to zero.
4. Desk-check:

sales	region	totalRegionSales
~~0~~	~~1~~	0
1000		1000
2000		3000
-1	2	
400		3400
500		3900
-3	3	

   The missing statement on Line 40 is `totalRegionSales = 0;`. Without this statement, the program is adding both regions' sales to the accumulator. As a result, it's displaying $3000 as the total sales for Region 1 (which is correct) and $3900 as the total sales for Region 2 (which is not correct). The program should be accumulating each region's sales separately and then displaying $3000 as the total sales for Region 1 and $900 as the total sales for Region 2. The `totalRegionSales = 0;` statement is needed to reset the accumulator to 0 before adding the next region's sales to it. You can modify the comment on Line 38 as follows: `//update the counter and reset the accumulator`.
5. Desk-check:

sales	region	totalRegionSales
~~0~~	~~1~~	0
1000		1000
2000		3000
-1	2	
0		
400		400
500		900
-3	3	

   The total sales for Region 1 are $3000. The total sales for Region 2 are $900.
6. The program displays $3000 as Region 1's total sales and $900 as Region 2's total sales.

7. The program displays $3000 as Region 1's total sales and $900 as Region 2's total sales.

```
//Lab8-1.cpp - displays each region's total sales
//Created/revised by <your name> on <current date>

#include <iostream>
using namespace std;

int main()
{
     //declare variables
     int sales            = 0;
     int totalRegionSales = 0;   //accumulator

     for (int region = 1; region < 3; region += 1)
     {
          //get current region's first sales amount
          cout << "First sales amount for Region "
               << region << ": ";
          cin >> sales;

          do //begin loop
          {
               //add the sales amount to the total
               //for the region
               totalRegionSales += sales;
               //get the next sales amount for the
               //current region
               cout << "Next sales amount for Region "
                    << region << ": ";
               cin >> sales;
          }    while (sales > 0);

          //display the current region's total sales
          cout << "****************Region " << region
               << " sales: $" << totalRegionSales
               << endl << endl;
          //reset the accumulator
          totalRegionSales = 0;
     }    //end for

     system("pause");
     return 0;
}    //end of main function
```



LAB 8-2 Plan and Create

No answer required.



LAB 8-3 Modify

```
//Lab8-3.cpp - displays a multiplication table
//Created/revised by <your name> on <current date>

#include <iostream>
using namespace std;

int main()
{
     //declare variables
     int multiplicand = 0;
     int product      = 0;

     cout << "Multiplicand (negative number to end): ";
     cin >> multiplicand;

     do //begin loop
     {
          int multiplier = 1;
          do //begin loop
          {
               product = multiplicand * multiplier;
               cout << multiplicand << " * "
                    << multiplier << " = "
                    << product << endl;
               multiplier += 1;
          }    while (multiplier < 10);

          cout << endl;
          cout << "Multiplicand (negative number to end): ";
          cin >> multiplicand;
     }    while (multiplicand >= 0);

     system("pause");
     return 0;
}    //end of main function
```



LAB 8-4 Desk-Check

Desk-check:

number	x
1	1
	2
	3
	4
	5
2	1
	2
	3
	4
	5
3	

The code will display the following:

1 2 3 4 5

2 3 4 5 6



LAB 8-5 Debug

To debug the program, cut the `//update the month counter` comment and `month += 1;` statement from the nested loop and paste them below the `cout << endl;` statement in the outer loop. Also, change the `totalSales += totalSales + sales;` statement to either `totalSales = totalSales + sales;` or `totalSales += sales;`.

Answers to Chapter 9 Mini-Quizzes

Mini-Quiz 9-1

1. b. `sqrt(16.0)`
2. c. `25 + rand() % (50 - 25 + 1)`
3. d. none of the above
4. a. `#include <ctime>`

Mini-Quiz 9-2

1. d. all of the above
2. a. `double getArea()`
3. `double getGrossPay (int hours, double rate)`
4. `return gross;`

Mini-Quiz 9-3

1. b. `area = getArea();`
2. c. `double getArea();`

3. `cout << getArea();`

4. `double getGrossPay(int hours, double rate);` or `double getGrossPay(int, double);`

5. `weekGross = getGrossPay(40, payRate);`

Mini-Quiz 9-4

1. b. False

2. b. False

3. a. True

4. When the variable appears in a statement in the program, the computer uses the location of the statement to determine which variable to use.

Answers to Chapter 9 Labs



LAB 9-1 Stop and Analyze

1. The `#include <ctime>` instruction on Line 5 is necessary because the program uses the `time` function.
2. The `srand(static_cast<int>(time(0)));` statement on Line 15 uses the `srand` and `time` functions to initialize the random number generator.
3. The number 4 will be assigned to the `randomNumber` variable.
4. No answer required.
5.

```
//Lab9-1.cpp - simulates a number guessing game
//Created/revised by <your name> on <current date>

#include <iostream>
#include <ctime>
using namespace std;

int main()
{
   //declare variables
   int randomNumber     = 0;
   int numberGuess      = 0;
   int incorrectGuesses = 0;
   char moreGuesses     = 'Y';
```

```
   //generate a random number from 1 through 10
   srand(static_cast<int>(time(0)));
   randomNumber = 1 + rand() % (10 - 1 + 1);

   //get first number guess from user
   cout << "Guess a number from 1 through 10: ";
   cin >> numberGuess;

   while (moreGuesses == 'Y')
   {
      if (numberGuess != randomNumber)
      {
         incorrectGuesses += 1;
         if (incorrectGuesses < 4)
         {
            cout << "Sorry, guess again: ";
            cin >> numberGuess;
         }
         else
         {
            cout << endl << "Sorry, the number is "
            << randomNumber << "." << endl;
            moreGuesses = 'N';
         }  //end if
      }
      else
      {
         moreGuesses = 'N';
         cout << endl << "Yes, the number is "
         << randomNumber << "." << endl;
      }  //end if

   }  //end while

   system("pause");
   return 0;
}  //end of main function
```

6.

```
//Lab9-1.cpp - simulates a number guessing game
//Created/revised by <your name> on <current date>

#include <iostream>
#include <ctime>
using namespace std;

//function prototype
int getRandomNumber();

int main()
{
   //declare variables
   int randomNumber     = 0;
   int numberGuess      = 0;
   int incorrectGuesses = 0;
   char moreGuesses     = 'Y';
```

```
   //generate a random number from 1 through 10
   srand(static_cast<int>(time(0)));
   randomNumber = getRandomNumber();

   //get first number guess from user
   cout << "Guess a number from 1 through 10: ";
   cin >> numberGuess;

   while (moreGuesses == 'Y')
   {
      if (numberGuess != randomNumber)
      {
         incorrectGuesses += 1;
         if (incorrectGuesses < 4)
         {
            cout << "Sorry, guess again: ";
            cin >> numberGuess;
         }
         else
         {
            cout << endl << "Sorry, the number is "
            << randomNumber << "." << endl;
            moreGuesses = 'N';
         }  //end if
      }
      else
      {
         moreGuesses = 'N';
         cout << endl << "Yes, the number is "
         << randomNumber << "." << endl;
      }  //end if

   }  //end while

   system("pause");
   return 0;
}  //end of main function

//*****function definitions*****
int getRandomNumber()
{
   int randInteger = 0;
   //generate random integer from 1 through 10
   randInteger = 1 + rand() % (10 - 1 + 1);
   return randInteger;
}  //end of getRandomNumber function
```

7.

```
//Lab9-1.cpp - simulates a number guessing game
//Created/revised by <your name> on <current date>

#include <iostream>
#include <ctime>
using namespace std;
```

```
//function prototype
int getRandomNumber(int lower, int upper);

int main()
{
   //declare variables
   int randomNumber     = 0;
   int numberGuess      = 0;
   int incorrectGuesses = 0;
   char moreGuesses     = 'Y';
   int smallest         = 0;
   int largest          = 0;

   cout << "Smallest integer: ";
   cin >> smallest;
   cout << "Largest integer: ";
   cin >> largest;
   cout << endl;

   //generate a random number from smallest through largest
   srand(static_cast<int>(time(0)));
   randomNumber = getRandomNumber(smallest, largest);

   //get first number guess from user
   cout << "Guess a number from " << smallest << " through "
   << largest << ": ";
   cin >> numberGuess;

   while (moreGuesses == 'Y')
   {
      if (numberGuess != randomNumber)
      {
         incorrectGuesses += 1;
         if (incorrectGuesses < 4)
         {
            cout << "Sorry, guess again: ";
            cin >> numberGuess;
         }
         else
         {
            cout << endl << "Sorry, the number is "
            << randomNumber << "." << endl;
            moreGuesses = 'N';
         }  //end if
      }
      else
      {
         moreGuesses = 'N';
         cout << endl << "Yes, the number is "
         << randomNumber << "." << endl;
      }  //end if

   }  //end while

   system("pause");
```

```
   return 0;
}  //end of main function

//*****function definitions*****
int getRandomNumber(int lower, int upper)
{
   int randInteger = 0;
   //generate random integer from lower through upper
   randInteger = lower + rand() % (upper - lower + 1);
   return randInteger;
}  //end of getRandomNumber function
```



LAB 9-2 Plan and Create

No answer required.



LAB 9-3 Modify

```
//Lab9-3.cpp - displays two monthly car payments
//Created/revised by <your name> on <current date>
#include <iostream>
#include <cmath>
#include <iomanip>
using namespace std;

//function prototype
double getPayment(int, double, int);

int main()
{
   //declare variables
   int carPrice         = 0;
   int rebate           = 0;
   double creditRate    = 0.0;
   double dealerRate    = 0.0;
   int term             = 0;
   double creditPayment = 0.0;
   double dealerPayment = 0.0;
   char again = 'Y';

   do
   {
      //get input items
      cout << "Car price (after any trade-in): ";
      cin >> carPrice;
      cout << "Rebate: ";
      cin >> rebate;
```

```
      cout << "Credit union rate: ";
      cin >> creditRate;
      cout << "Dealer rate: ";
      cin >> dealerRate;
      cout << "Term in years: ";
      cin >> term;

      //convert rates to decimal format, if necessary
      if (creditRate >= 1.0)
         creditRate /= 100;
      //end if
      if (dealerRate >= 1.0)
         dealerRate /= 100;
      //end if

      //call function to calculate payments
      creditPayment = getPayment(carPrice - rebate,
         creditRate / 12, term * 12);
      dealerPayment = getPayment(carPrice,
         dealerRate / 12, term * 12);

      //display payments
      cout << fixed << setprecision(2) << endl;
      cout << "Credit union payment: $"
         << creditPayment << endl;

      cout << "Dealer payment: $"
         << dealerPayment << endl;

      if (creditPayment < dealerPayment)
         cout << "Take the rebate and finance through the
credit union." << endl;
      else
         if (creditPayment > dealerPayment)
         cout << "Don't take the rebate. Finance through
the dealer." << endl;
         else
            cout << "You can finance through either one."
<< endl;
         //end if
      //end if

      cout << endl << "Make another calculation? (Y/N): ";
      cin >> again;
      cout << endl;
   }  while (toupper(again) == 'Y');

   system("pause");
   return 0;
}  //end of main function

//*****function definitions*****
double getPayment(int prin,
                  double monthRate,
                  int months)
```

```
{
   //calculates and returns a monthly payment
   double monthPay = 0.0;
   monthPay = prin * monthRate /
      (1 - pow(monthRate + 1, -months));
   return monthPay;
}  //end of getPayment function
```



LAB 9-4 Desk-Check

Desk-check:

balance	amount	transaction	another
0.0	0.0	-	Y
2000.0	400.0	W	y
1600.0	1200.0	W	y
2800.0	45.0	D	y
2755.0	55.0	D	y
2700.0	150.0	W	y
2550.0	15.0	W	y
2565.0	1050.0	W	y
1515.0		W	n
		R	
		K	
		W	
		W	
		d	
		D	
		W	
		W	

bal	type	amt	curBalance
2000.0	W	400.0	0.0
			1600.0
1600.0	D	1200.0	0.0
			2800.0
2800.0	W	45.0	0.0
			2755.0
2755.0	W	55.0	0.0
			2700.0
2700.0	W	150.0	0.0
			2550.0
2550.0	D	15.0	0.0
			2565.0
2565.0	W	1050.0	0.0
			1515.0

The code will display a current balance of $1515.00.



LAB 9-5 Debug

To debug the program, change the statement that calls the `getDepreciation` function to `depreciation = getDepreciation(cost, salvage, lifeYears);`.

Answers to Chapter 10 Mini-Quizzes

Mini-Quiz 10-1

1. `void`
2. `displayTaxes(federalTax, localTax);`
3. `void displayTaxes(double fedTax, double stateTax)`
4. b. False

Mini-Quiz 10-2

1. a. `void getInput(double &hours, double &rate)`
2. b. `getInput(hoursWkd, payRate);`
3. `void getInput(double &hours, double &rate);` or `void getInput(double &, double &);`
4. a. True

Mini-Quiz 10-3

1. `void calcTaxes(double pay, double &fedTax, double &stateTax)`
2. `calcTaxes(gross, federal, state);`
3. `void calcTaxes(double pay, double &fedTax, double &stateTax);` or `void calcTaxes(double, double &, double &);`
4. b. False

Answers to Chapter 10 Labs



LAB 10-1 Stop and Analyze

1. The function prototype on Lines 11 through 13 and the function header on Lines 77 through 79.
2. The `americanDols` and `conversionRate` variables are passed *by value* because the `convertDols` function needs to know their values but does not need to change their contents. The `convertedCurrency` variable is passed by reference because the `convertDols` function needs to store a value (the converted dollars) inside the variable.
3. The `displayMenu` function is a void function because it does not return a value after completing its task.
4. Line 35 contains the priming read. Line 61 contains the update read.
5. You would need to modify the function as shown here:

```
double convertDols(double dollars, double convertRate)
{
      double converted = 0.0;
      converted = dollars * convertRate;
      return converted;
}     //end of convertDols function
```

   You also need to change the function prototype on Lines 11 through 13 to `double convertDols(double dollars, double convertRate);` and change the function call on Lines 52 through 54 to `convertedCurrency = convertDols(americanDollars, conversionRate);`.
6. No answer required.
7. 31.53
8. 135.58
9.

```
//Lab10-1.cpp - Converts American dollars to
//British pounds, Mexican pesos, or Japanese yen
//Created/revised by <your name> on <current date>

#include <iostream>
#include <iomanip>
using namespace std;

//function prototypes
void displayMenu();
void convertDols(double dollars,
                 double convertRate,
                 double &converted);
void assignRate(int choice, double &convertRate);
```

```
int main()
{
     //declare variables
     int menuChoice           = 0;
     double americanDollars   = 0.0;
     double conversionRate    = 0.0;
     double convertedCurrency = 0.0;

     //display output in fixed-point notation
     //with two decimal places
     cout << fixed << setprecision(2);

     //get menu choice
     displayMenu();
     cout << "Enter 1, 2, 3, or 4: ";
     cin >> menuChoice;

     while (menuChoice > 0 && menuChoice < 4)
     {
          //get dollars to convert
          cout << "Number of American dollars: ";
          cin >> americanDollars;

          //assign rate
          assignRate(menuChoice, conversionRate);

          convertDols(americanDollars,
                      conversionRate,
                      convertedCurrency);
          cout << "-->" << convertedCurrency
               << endl << endl;

          //get menu choice
          displayMenu();
          cout << "Enter 1, 2, 3, or 4: ";
          cin >> menuChoice;
     }    //end while

     system("pause");
     return 0;
}    //end of main function

//*****function definitions*****
void displayMenu()
{
     cout << "1 British pounds" << endl;
     cout << "2 Mexican pesos" << endl;
     cout << "3 Japanese yen" << endl;
     cout << "4 Stop program" << endl;
}    //end of displayMenu function

void convertDols(double dollars,
                 double convertRate,
                 double &converted)
{
     converted = dollars * convertRate;
}    //end of convertDols function
```

```
void assignRate(int choice, double &convertRate)
{
     //declare constants
     const double BRITISH_RATE  = .630676;
     const double MEXICAN_RATE  = 13.5584;
     const double JAPANESE_RATE = 88.7626;
     if (choice == 1)
          convertRate = BRITISH_RATE;
     else if (choice == 2)
          convertRate = MEXICAN_RATE;
     else
          convertRate = JAPANESE_RATE;
     //end if
}    //end of assignRate function
```

10.

```
//Lab10-1.cpp - Converts American dollars to
//British pounds, Mexican pesos, or Japanese yen
//Created/revised by <your name> on <current date>

#include <iostream>
#include <iomanip>
using namespace std;

//function prototypes
void displayMenu();
double convertDols(double dollars,
                   double convertRate);
void assignRate(int choice, double &convertRate);

int main()
{
     //declare variables
     int menuChoice           = 0;
     double americanDollars   = 0.0;
     double conversionRate    = 0.0;
     double convertedCurrency = 0.0;

     //display output in fixed-point notation
     //with two decimal places
     cout << fixed << setprecision(2);

     //get menu choice
     displayMenu();
     cout << "Enter 1, 2, 3, or 4: ";
     cin >> menuChoice;

     while (menuChoice > 0 && menuChoice < 4)
     {
          //get dollars to convert
          cout << "Number of American dollars: ";
          cin >> americanDollars;

          //assign rate
          assignRate(menuChoice, conversionRate);
```

```
          convertedCurrency =
               convertDols(americanDollars, conversionRate);
          cout << "-->" << convertedCurrency
               << endl << endl;

          //get menu choice
          displayMenu();
          cout << "Enter 1, 2, 3, or 4: ";
          cin >> menuChoice;
     }    //end while

     system("pause");
     return 0;
}    //end of main function

//*****function definitions*****
void displayMenu()
{
     cout << "1 British pounds" << endl;
     cout << "2 Mexican pesos" << endl;
     cout << "3 Japanese yen" << endl;
     cout << "4 Stop program" << endl;
}    //end of displayMenu function

double convertDols(double dollars,
                   double convertRate)
{
     double converted = 0.0;
     converted = dollars * convertRate;
     return converted;
}    //end of convertDols function

void assignRate(int choice, double &convertRate)
{
     //declare constants
     const double BRITISH_RATE = .630676;
     const double MEXICAN_RATE = 13.5584;
     const double JAPANESE_RATE = 88.7626;

     if (choice == 1)
          convertRate = BRITISH_RATE;
     else if (choice == 2)
          convertRate = MEXICAN_RATE;
     else
          convertRate = JAPANESE_RATE;
     //end if
}    //end of assignRate function
```



LAB 10-2 Plan and Create

No answer required.



LAB 10-3 Modify

```
//Lab10-3.cpp - displays the number of units of
//electricity used and the total charge
//Created/revised by <your name> on <current date>

#include <iostream>
#include <iomanip>
using namespace std;

//function prototypes
void getInput(int &newReading, int &oldReading);
void getUnits(int curRead, int prevRead, int &numUnits);
double getTotal(int numUnits, double chgPerUnit, double
&totChg);
void displayBill(int used, double charge);

int main()
{
     //declare constant and variables
     const double UNIT_CHG = .09;
     int current           = 0;
     int previous          = 0;
     int units             = 0;
     double total          = 0.0;

     cout << fixed << setprecision(2);

     //call functions
     getInput(current, previous);
     getUnits(current, previous, units);
     total = getTotal(units, UNIT_CHG, total);
     displayBill(units, total);

     system("pause");
     return 0;
}    //end of main function

//*****function definitions*****
void getInput(int &newReading, int &oldReading)
{
     cout << "Current reading: ";
     cin >> newReading;
     cout << "Previous reading: ";
     cin >> oldReading;
}    //end of getInput function

void getUnits(int curRead, int prevRead, int &numUnits)
{
     numUnits = curRead - prevRead;
}    //end of getUnits function

double getTotal(int numUnits, double chgPerUnit, double
&totChg)
{
          totChg = numUnits * chgPerUnit;
```

```
          return totChg;
}    //end of getTotal function

void displayBill(int used, double charge)
{
     cout << "Units used: " << used << endl;
     cout << "Total charge: $" << charge << endl;
}    //end of displayBill function
```



LAB 10-4 Desk-Check

test1 (main)	test2 (main)	~~avg (calcAvg)~~ average (main)	num1 (calcAvg)	num2 (calcAvg)
~~0.0~~	~~0.0~~	~~0.0~~	~~78.0~~	~~85.0~~
~~78.0~~	~~85.0~~	~~81.5~~	~~45.0~~	~~93.0~~
~~45.0~~	~~93.0~~	~~69.0~~	~~87.0~~	~~98.0~~
~~87.0~~	~~98.0~~	~~92.5~~	54.0	32.0
54.0	32.0	43.0		



LAB 10-5 Debug

To debug the program, change the function prototype to `void assignGrade(int pointsEarned, char &letter);` and change the function header to `void assignGrade(int pointsEarned, char &letter)`.

Answers to Chapter 11 Mini-Quizzes

Mini-Quiz 11-1

1. a. `int quantities[20] = {0};`
2. `quantities[0]`
3. `quantities[19]`
4. `quantities[3] = 7;`
5. c. `total = getTotal(quantities, 20);`

Mini-Quiz 11-2

1. c. `total += orders[2];`
2. c. `if (orders[3] > 25)`
3. `bonus = sales[0] * .15;`
4. c. `if (sub >= 0 && sub < 10)`
5. a. `while (x < 20)`

Mini-Quiz 11-3

1. `if (prices[x] < lowest)`
2. sorting
3.
```
for (int x = 0; x < 10; x += 1)
        orders[x] -= 3;
//end for
```

Answers to Chapter 11 Labs



LAB 11-1 Stop and Analyze

1. The `domestic` and `international` arrays are parallel arrays because the elements in one array are related by their subscripts to the elements in the other array. For example, the first element in both arrays contains the sales made in January. The second element contains the February sales and so on.
2. The `domestic[1]` element contains 45000.
3. The total company sales made in February can be calculated by adding the contents of the `domestic[1]` element to the contents of the `international[1]` element.
4. The highest subscript in the `international` array is 5.
5. The assignment statement would need to be changed to `totalSales += domestic[x - 1] + international[x - 1];`.
6. No answer required.

7.

```
//Lab11-1.cpp - calculates the total domestic sales,
//total international sales, and total sales
//Created/revised by <your name> on <current date>

#include <iostream>
using namespace std;

int main()
{
     //declare arrays and variable
     int domestic[6]        = {12000, 45000, 32000,
                               67000, 24000, 55000};
     int international[6]   = {10000, 56000, 42000,
                               23000, 12000, 34000};
     int totalSales         = 0; //accumulator
     int totalDomestic      = 0; //accumulator
     int totalInternational = 0; //accumulator

     //accumulate sales
     for (int x = 0; x < 6; x += 1)
     {
          totalDomestic += domestic[x];
          totalInternational += international[x];
          totalSales += domestic[x] + international[x];
     }    //end for

     //display total domestic sales, total
     //international sales, and total sales
     cout << "Total domestic sales: $"
          << totalDomestic << endl;
     cout << "Total international sales: $"
          << totalInternational << endl;
     cout << "Total sales: $" << totalSales << endl;

     system("pause");
     return 0;
}    //end of main function
```

8.

```
//Lab11-1.cpp - calculates the total domestic sales,
//total international sales, total sales, and
//total sales made in each month
//Created/revised by <your name> on <current date>

#include <iostream>
using namespace std;

int main()
{
     //declare arrays and variable
     int domestic[6] = {12000, 45000, 32000,
                        67000, 24000, 55000};
     int international[6] = {10000, 56000, 42000,
                             23000, 12000, 34000};
```

```
     int totalSales         = 0; //accumulator
     int totalDomestic      = 0; //accumulator
     int totalInternational = 0; //accumulator
     int monthSales[6]      = {0};

     //accumulate sales
     for (int x = 0; x < 6; x += 1)
     {
          totalDomestic += domestic[x];
          totalInternational += international[x];
          totalSales += domestic[x] + international[x];
          monthSales[x] = domestic[x] + international[x];
     }    //end for

     //display total domestic sales, total
     //international sales, and total sales
     cout << "Total domestic sales: $"
          << totalDomestic << endl;
     cout << "Total international sales: $"
          << totalInternational << endl;
     cout << "Total sales: $" << totalSales << endl;

     //display total sales made in each month
     for (int x = 0; x < 6; x += 1)
          cout << "Month " << x + 1 << " sales: $"
          << monthSales[x] << endl;
     //end for

     system("pause");
     return 0;
}    //end of main function
```



LAB 11-2 Plan and Create

No answer required.



LAB 11-3 Modify

```
//Lab11-3.cpp
//Stores monthly rainfall amounts in an array
//Displays the monthly rainfall amounts or the
//total annual rainfall amount
//Created/revised by <your name> on <current date>

#include <iostream>
using namespace std;

//function prototypes
void displayMonthly(double rain[], int numElements);
double getTotal(double rain[], int numElements);
```

```
int main()
{
   //declare array and variables
   double rainfall[12]  = {0.0};
   int choice           = 0;
   double totalRainfall = 0.0;

   //get rainfall amounts
   for (int x = 0; x < 12; x += 1)
   {
      cout << "Enter rainfall for month " << x + 1 << ": ";
      cin >> rainfall[x];
   }  //end for

   do
   {
      //display menu and get menu choice
      cout << endl;
      cout << "1 Display monthly amounts" << endl;
      cout << "2 Display total amount" << endl;
      cout << "3 End program" << endl;
      cout << "Enter your choice: ";
      cin >> choice;

      if (choice != 1 && choice != 2 && choice != 3)
         cout << "Invalid choice" << endl << endl;
      else
      {
         //call appropriate function or end program
         if (choice == 1)
            displayMonthly(rainfall, 12);
         else
            if (choice == 2)
            {
               totalRainfall = getTotal(rainfall, 12);
               cout << "Total rainfall: "
               << totalRainfall << endl;
            }  //end if
         //end if
      } //end if
   } while (choice != 3);

   system("pause");
   return 0;
}  //end of main function

//*****function definitions*****
void displayMonthly(double rain[], int numElements)
{
   cout << "Monthly rainfall amounts:" << endl;
   for (int x = 0; x < 12; x += 1)
      cout << rain[x] << endl;
   //end for
} //end of displayMonthly function
```

```
double getTotal(double rain[], int numElements)
{
   double total = 0.0;
   for (int x = 0; x < 12; x += 1)
      total = total + rain[x];
   //end for
   return total;
}  //end of getTotal function
```



LAB 11-4 Desk-Check

Desk-check:

midterms[0]	midterms[1]	midterms[2]	midterms[3]	midterms[4]
~~0.0~~	~~0.0~~	~~0.0~~	~~0.0~~	~~0.0~~
90.0	88.0	~~77~~.0	85.0	45.0

finals[0]	finals[1]	finals[2]	finals[3]	finals[4]
~~0.0~~	~~0.0~~	~~0.0~~	0.0	~~0.0~~
100.0	68.0	~~75~~.0	85.0	32.0

averages[0]	averages[1]	averages[2]	averages[3]	averages[4]
~~0.0~~	~~0.0~~	~~0.0~~	~~0.0~~	~~0.0~~
95.0	~~78~~.0	~~76~~.0	85.0	38.5

x
~~0~~
~~1~~
~~2~~
~~3~~
~~4~~
5

The `for` loop will display the following:
Student 1 average: 95
Student 2 average: 78
Student 3 average: 76
Student 4 average: 85
Student 5 average: 38.5



LAB 11-5 Debug

To debug the program, change the `increase += quantities[x];` statement in the `for` loop to `quantities[x] += increase;`.

Answers to Chapter 12 Mini-Quizzes

Mini-Quiz 12-1

1. b. `int quantities[4][2] = {0};`
2. 20
3. `quantities[0][0]`
4. `quantities[3][1]`
5. `quantities[0][1] = 5;`

Mini-Quiz 12-2

1. d. `total += purchases[1][0];`
2. c. `if (scores[2][3] > 25)`
3. `bonus = sales[0][1] * .15;`
4. a. `if (row >= 0 && row < 10)`

Answers to Chapter 12 Labs



LAB 12-1 Stop and Analyze

1. 34000
2. The total company sales made in February can be calculated by adding the contents of the `company[0][1]` element to the contents of the `company[1][1]` element.
3. The highest row subscript in the `company` array is 1. The highest column subscript is 5.
4. The January international sales are stored in the `company[1][0]` element.
5.
```
int location = 0;
while (location < 2)
   for (int month = 0; month < 6; month += 1)
      companySales += company[location[month];
   //end for
   location += 1;
//end while
```

6. The assignment statement would need to be changed to `companySales += company[location[month - 1];`.

7. No answer required.

8.

```
//Lab12-1.cpp - calculates the total domestic sales,
//total international sales, and total company sales
//Created/revised by <your name> on <current date>

#include <iostream>
using namespace std;

int main()
{
     //declare arrays and variable
     int company[2][6] = {{12000, 45000, 32000,
                           67000, 24000, 55000},
                          {10000, 56000, 42000,
                           23000, 12000, 34000}};
     int companySales       = 0;
     int domesticSales      = 0;
     int internationalSales = 0;

     //accumulate sales
     for (int month = 0; month < 6; month += 1)
     {
          domesticSales += company[0][month];
          internationalSales += company[1][month];
     }    //end for
     companySales = domesticSales + internationalSales;

     //display total sales
     cout << "Domestic sales: $" << domesticSales << endl;
     cout << "International sales: $" << internationalSales
     << endl;
     cout << "Company sales: $" << companySales << endl;

     system("pause");
     return 0;
}    //end of main function
```

9.

```
//Lab12-1.cpp - calculates the total domestic sales,
//total international sales, and total company sales
//Created/revised by <your name> on <current date>

#include <iostream>
using namespace std;

int main()
{
   //declare arrays and variable
```

```
   int company[2][6] = {{12000, 45000, 32000,
                         67000, 24000, 55000},
                         {10000, 56000, 42000,
                         23000, 12000, 34000}};
   int monthSales[6]      = {0};
   int companySales       = 0;
   int domesticSales      = 0;
   int internationalSales = 0;

   //accumulate sales
   for (int month = 0; month < 6; month += 1)
   {
      domesticSales += company[0][month];
      internationalSales += company[1][month];
      monthSales[month] = company[0][month] + company[1]
[month];
   }  //end for
   companySales = domesticSales + internationalSales;

   //display total sales
   cout << "Domestic sales: $" << domesticSales << endl;
   cout << "International sales: $" << internationalSales
<< endl;
   cout << "Company sales: $" << companySales << endl;

   //display monthly sales
   for (int x = 0; x < 6; x += 1)
      cout << "Month " << x + 1 << " sales: $"
      << monthSales[x] << endl;
   //end for

   system("pause");
   return 0;
}  //end of main function
```



LAB 12-2 Plan and Create

No answer required.



LAB 12-3 Modify

```
//Lab12-3.cpp - displays the shipping charge
//Created/revised by <your name> on <current date>
#include <iostream>
using namespace std;

int main()
{
```

```
    //declare array and variables
    int shipCharges[3][2] = {{101, 0},
                             {51, 10},
                             {1, 20}};
    int numOrdered = 0;
    int rowSub     = 0;
    char found     = 'N';

    //enter the number ordered
    cout << "Number ordered ";
    cout << "(negative number or 0 to end): ";
    cin >> numOrdered;

    while (numOrdered > 0 && numOrdered <= 999999)
    {
         //search array
         rowSub = 0;
         while (rowSub < 3 && found == 'N')
              if (numOrdered >= shipCharges[rowSub][0])
                   found = 'Y';
              else
                   rowSub += 1;
              //end if
         //end while

         //display shipping charge
         cout << "Shipping charge for a quantity of "
              << numOrdered << " is $"
              << shipCharges[rowSub][1] << endl << endl;

         //enter the number ordered
         cout << "Number ordered ";
         cout << "(negative number or 0 to end): ";
         cin >> numOrdered;
         found = 'N';
    }    //end while

    system("pause");
    return 0;
}    //end of main function
```



LAB 12-4 Desk-Check

sales[0][0]	sales[0][1]
~~0.0~~	~~0.0~~
90.0	88.0

sales[1][0]	sales[1][1]
~~0.0~~	~~0.0~~
100.0	68.0

sales[2][0]	sales[2][1]
0.0	0.0
95.0	78.0

total	store	book
0.0	0	0
1200.33	1	1
3551.08	8	2
7228.88	3	0
9684.93		1
10435.6		2
11781.59		0
		1
		2



LAB 12-5 Debug

To debug the program, change the `numbers[row][0] = counter * counter;` statement in the first `for` loop to `numbers[row][1] = counter * counter;` and then enter the `counter += 1;` (or `counter = counter + 1;`) statement below the `numbers[row][1] = counter * counter;` statement.

Answers to Chapter 13 Mini-Quizzes

Mini-Quiz 13-1

1. c. `const string FIRST_PRES = "George Washington";`
2. b. `string country = "";`
3. a. `getline(cin, streetAddress, '\n');`
4. c. `cin.ignore(10, '\n');`

Mini-Quiz 13-2

1. a. `while (employee.length() > 20)`
2. c. `if (code.length() == 7)`
3. d. both a and b
4. `cout << college.substr(college.length() - 1);`

Mini-Quiz 13-3

1. a. `location = cityState.find(",", 0);`
2. 13
3. d. all of the above

Mini-Quiz 13-4

1. d. none of the above
2. a. `sentence = sentence + temp.assign(4, '!');`
3. b. `areaCode = "(" + areaCode + ")";`

Answers to Chapter 13 Labs



LAB 13-1 Stop and Analyze

1. The purpose of the loop is to access each character in the phone variable, one character at a time, in order to remove any parentheses or hyphens.
2. The statement in Line 26 assigns the current character from the phone variable to the `currentChar` variable .
3. The selection structure in Lines 27 through 36 compares the current character in the phone variable with the opening and closing parentheses and the hyphen. If the current character is an opening or closing parenthesis or a hyphen, the statement in Line 31 removes the current character from the phone variable. The statement in Line 32 then subtracts the number 1 from the `numChars` variable, which keeps track of the number of characters in the phone variable. If the current character is not an opening or closing parenthesis or a hyphen, the statement in Line 35 adds 1 to the contents of the `subscript` variable, which allows the loop to access the next character in the phone variable.
4. Before the loop in the program is processed, the statement in Line 21 assigns the number of characters stored in the phone variable to the `numChars` variable. If the phone variable contains the eight characters 111-2222, the statement assigns the number 8 to the `numChars` variable. When the statement in Line 31 removes a character from the phone variable, the variable contains one less character. For example, if the statement in Line 31 removes the hyphen from the 111-2222 stored in the phone variable, the variable contains seven characters

rather than eight characters. If you do not subtract the number 1 from the `numChars` variable, the loop will attempt to access the eighth character in the `phone` variable, even though the variable now contains only seven characters.

5. Assume that the `phone` variable contains (500)333-4444. The statement in Line 26 assigns the opening parenthesis, whose subscript is 0, to the `currentChar` variable. When the statement in Line 31 removes the opening parenthesis from the `phone` variable, the number 5 becomes the first character in the variable and, therefore, it has a subscript of 0. In other words, when you remove a character from a variable, the next character in the variable assumes the same subscript. However, when a character is not removed from the variable, you need to update the subscript to access the next character in the variable.
6. No answer required.
7. Change the `phone.erase(subscript, 1);` statement in Line 31 to `phone.replace(subscript, 1, "");`.



LAB 13-2 Plan and Create

No answer required.



LAB 13-3 Modify

```
//Lab13-3.cpp - simulates the Hangman game
//Created/revised by <your name> on <current date>
#include <iostream>
#include <string>
using namespace std;

int main()
{
    //declare variables
    string origWord    = "";
    string letter      = "";
    char dashReplaced  = 'N';
    char gameOver      = 'N';
    int numIncorrect   = 0;
    string displayWord = "";
    int numChars       = 0;

    //get original word
    cout << "Enter a word in uppercase: ";
    getline(cin, origWord);
    numChars = origWord.length();
    displayWord.assign(numChars, '-');
```

```
//clear the screen
system("cls");

//start guessing
cout << "Guess this word: " <<
    displayWord << endl;
while (gameOver == 'N')
{
    cout << "Enter an uppercase letter: ";
    cin >> letter;

    //search for the letter in the original word
    for (int x = 0; x < numChars; x += 1)
    {
        //if the current character matches
        //the letter, replace the corresponding
        //dash in the displayWord variable and then
        //set the dashReplaced variable to 'Y'
        if (origWord.substr(x, 1) == letter)
        {
            displayWord.replace(x, 1, letter);
            dashReplaced = 'Y';
        }   //end if
    }   //end for

    //if a dash was replaced, check whether the
    //displayWord variable contains any dashes
    if (dashReplaced == 'Y')
    {
        //if the displayWord variable does not
        //contain any dashes, the game is over
        if (displayWord.find("-", 0) == -1)
        {
            gameOver = 'Y';
            cout << endl << "Yes, the word is "
                << origWord << endl;
            cout << "Great guessing!" << endl;
        }
        else //otherwise, continue guessing
        {
            cout << endl << "Guess this word: "
                << displayWord << endl;
            dashReplaced = 'N';
        }   //end if
    }
    else //processed when dashReplaced contains 'N'
    {
        //add 1 to the number of incorrect guesses
        numIncorrect += 1;
        //if the number of incorrect guesses is 10,
        //the game is over
        if (numIncorrect == 10)
        {
            gameOver = 'Y';
```

```
                cout << endl << "Sorry, the word is "
                    << origWord << endl;
            }  //end if
        }  //end if
    }  //end while

    system("pause");
    return 0;
}   //end of main function
```



LAB 13-4 Desk-Check

Desk-check:

message	subMessage1	subMessage2
~~praogxwrazingmun~~		
~~praograzingmun~~	~~praograzing~~	!!!!!
~~praograzingismun~~	~~praogramming~~	~~isfun!!!!!~~
~~praograzingisfun~~	~~prgramming~~	~~isfun!!!!!~~
Programming is fun!!!!!	~~Prgramming~~	is fun!!!!!
	Programming	

The code will display the following: Message: Programming is fun!!!!!



LAB 13-5 Debug

To debug the program, change the `cout << message.substr(x) << endl;` statement in the `for` loop to `cout << message.substr(x, 1) << endl;`.

Answers to Chapter 14 Mini-Quizzes

Mini-Quiz 14-1

1. b. `#include <fstream>`
2. d. `ios::in`
3. `ofstream outAlbums;`
4. c. `outAlbums.open("mine.txt", ios::app);`

Mini-Quiz 14-2

1. the Boolean value `false`
2. c. `outInv << quantity << endl;`
3. d. `outFile << score1 << '#' << score2 << endl;`
4. d. `inInv >> number;`

Mini-Quiz 14-3

1. c. `while (!inInv.eof())`
2. the Boolean value `false`
3. `outInv.close();`

Answers to Chapter 14 Labs



LAB 14-1 Stop and Analyze

1. The instruction on Line 5 is necessary because the program uses the `cin` and `cout` objects. The instruction on Line 6 is necessary because the program uses the `string` class. The instruction on Line 7 is necessary because the program uses the `ofstream` class.
2. Each record contains two fields: the movie title and the year the movie was released.
3. The movies.txt file will contain only the two records written during the second run of the program. This is because the program opens the movies.txt file for output, which means the file's contents will be erased each time the program is run.
4. To save the previous records, you will need to open the file for append by changing the *mode* in the `open` function in Line 18 to `ios::app`.
5. `if (outFile.is_open() == true)`
6. The purpose of the # character is to separate the movie title field from the year released field.
7. The statement in Line 42 closes the output file. Neglecting to close a file can result in a loss of data.
8. No answer required.

9. No answer required.
10. No answer required.
11. Change the `outFile.open("movies.txt", ios::out);` statement in Line 18 to `outFile.open("movies.txt", ios::app);`.



LAB 14-2 Plan and Create

No answer required.



LAB 14-3 Modify

```
//Lab14-3.cpp - saves records to a sequential access
//file and also calculates and displays the total
//of the sales amounts stored in the file
//Created/revised by <your name> on <current date>
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

//function prototypes
int getChoice();
void addRecords();
void displayRecords();
void displayTotal();
void displayAvg();

int main()
{
     int choice = 0;
     do
     {
          //get user's menu choice
          choice = getChoice();
          if (choice == 1)
               addRecords();
          else
               if (choice == 2)
                    displayRecords();
               else
                    if (choice == 3)
                         displayTotal();
                    else
                         if (choice == 4)
                              displayAvg();
                         //end if
                    //end if
```

```
                    //end if
               //end if
          }    while (choice != 5);

               system("pause");
               return 0;
}    //end of main function

//*****function definitions*****
int getChoice()
{
     //displays menu and returns choice

     int menuChoice = 0;
     cout << endl << "Menu Options" << endl;
     cout << "1 Add Records" << endl;
     cout << "2 Display Records" << endl;
     cout << "3 Display Total Sales" << endl;
     cout << "4 Display Average Sales" << endl;
     cout << "5 Exit the program" << endl;
     cout << "Choice (1, 2, 3, 4, or 5)? ";
     cin >> menuChoice;
     cin.ignore(100, '\n');
     cout << endl;
     return menuChoice;
}    //end of getChoice function

void addRecords()
{
     //saves records to a sequential access file

     string name = "";
     int sales   = 0;
     ofstream outFile;

     //open file for append
     outFile.open("sales.txt", ios::app);

     //if the open was successful, get the
     //salesperson's name and sales amount and
     //then write the information to the file;
     //otherwise, display an error message
     if (outFile.is_open())
     {
          cout << "Salesperson's name (X to stop): ";
          getline(cin, name);

          while (name != "X" && name != "x")
          {
               cout << "Sales: ";
               cin >> sales;
               cin.ignore(100, '\n');

               outFile << name << '#'     << sales << endl;

               cout << "Salesperson's name "
                    << "(X to stop): ";
```

```
               getline(cin, name);
          }    //end while
          outFile.close();
     }
     else
          cout << "File could not be opened." << endl;
     //end if
}     //end of addRecords function

void displayRecords()
{
     //displays the contents of the sales.txt file

     string name = "";
     int sales   = 0;
     ifstream inFile;

     //open file for input
     inFile.open("sales.txt");

     //if the open was successful, read a
     //record and then display the record
     //otherwise, display an error message
     if (inFile.is_open())
     {
          getline(inFile, name, '#');
          inFile >> sales;
          inFile.ignore();

          while (!inFile.eof())
          {
               cout << name << "     $" << sales << endl;
               getline(inFile, name, '#');
               inFile >> sales;
               inFile.ignore();
     }    //end while
          inFile.close();
     }
     else
          cout << "File could not be opened." << endl;
     //end if
}    //end of displayRecords function

void displayTotal()
{
     //calculates and displays the total sales

     string name = "";
     int sales   = 0;
     int total   = 0;
     ifstream inFile;

     //open file for input
     inFile.open("sales.txt");

     //if the open was successful, read the
```

```
     //salesperson's name and sales amount, then add
     //the sales amount to the accumulator, and then
     //display the accumulator; otherwise, display
     //an error message
     if (inFile.is_open())
     {
          getline(inFile, name, '#');
          inFile >> sales;
          inFile.ignore();

          while (!inFile.eof())
          {
               total += sales;
               getline(inFile, name, '#');
               inFile >> sales;
               inFile.ignore();
     }    //end while
          inFile.close();
          cout << "Total sales $" << total
               << endl << endl;
     }
     else
          cout << "File could not be opened." << endl;
     //end if
}    //end of displayTotal function

void displayAvg()
{
     //calculates and displays the average sales

     string name     = "";
     int sales       = 0;
     int totalSales  = 0;
     int numSales    = 0;
     double avgSales = 0.0;

     ifstream inFile;

     //open file for input
     inFile.open("sales.txt");

     //if the open was successful, read the
     //salesperson's name and sales amount, then add
     //the sales amount to the accumulator and add 1
     //to the counter; otherwise, display an error message
     if (inFile.is_open())
     {
          getline(inFile, name, '#');
          inFile >> sales;
          inFile.ignore();

          while (!inFile.eof())
          {
               totalSales += sales;
               numSales += 1;
```

```
                getline(inFile, name, '#');
                inFile >> sales;
                inFile.ignore();
        }    //end while
             inFile.close();

             //calculate and display the average sales
             avgSales =
                  static_cast<double>(totalSales) / numSales;
             cout << "Average sales $" << avgSales
                  << endl << endl;
        }
        else
             cout << "File could not be opened." << endl;
        //end if
}    //end of displayTotal function
```



LAB 14-4 Desk-Check

Desk-check:

store1Sales	store2Sales	store1Total	store2Total
~~0~~	~~0~~	~~0~~	~~0~~
~~2000~~	~~4000~~	~~2000~~	~~4000~~
~~3500~~	~~4600~~	~~5500~~	~~8600~~
~~1250~~	~~2300~~	~~6750~~	~~10900~~
9300	8950	16050	19850

The code will display the following:
Store 1's total sales: $16050
Store 2's total sales: $19850



LAB 14-5 Debug

To debug the program, change the `outFile.open("records.txt", ios::in);` statement to either `outFile.open("records.txt", ios::out);` or `outFile.open("records.txt");`. In addition, add the `cin.ignore(100, '\n');` statement below the `cin >> num2;` statement.

APPENDIX B

C++ Keywords

```
abstract
and
and_eq
array
asm
auto
bitand
bitor
bool
break
case
catch
char
class
compl
const
const_cast
continue
default
delegate
delete
do
double
dynamic_cast
else
enum
event
explicit
export
extern
false
finally
float
for
friend
gcnew
generic
goto
if
initonly
inline
int
interface
interior_ptr
literal
long
mutable
namespace
new
not
not_eq
nullptr
operator
or
or_eq
private
property
protected
public
register
reinterpret_cast
return
safe_cast
sealed
short
signed
sizeof
static
static_cast
struct
switch
template
this
throw
true
try
typedef
typeid
typename
union
unsigned
using
virtual
void
volatile
wchar_t
while
xor
xor_eq
```

APPENDIX C

ASCII Codes

Character	ASCII	Binary
SPACE	32	00100000
!	33	00100001
“	34	00100010
#	35	00100011
$	36	00100100
%	37	00100101
&	38	00100110
’	39	00100111
(	40	00101000
)	41	00101001
*	42	00101010
+	43	00101011
‘	44	00101100
–	45	00101101
.	46	00101110
/	47	00101111
0	48	00110000
1	49	00110001
2	50	00110010
3	51	00110011
4	52	00110100
5	53	00110101
6	54	00110110
7	55	00110111
8	56	00111000
9	57	00111001
:	58	00111010
;	59	00111011
<	60	00111100
=	61	00111101
>	62	00111110
?	63	00111111
@	64	01000000
A	65	01000001
B	66	01000010
C	67	01000011
D	68	01000100
E	69	01000101
F	70	01000110
G	71	01000111
H	72	01001000
I	73	01001001
J	74	01001010
K	75	01001011
L	76	01001100
M	77	01001101
N	78	01001110
O	79	01001111
P	80	01010000
Q	81	01010001
R	82	01010010
S	83	01010011
T	84	01010100
U	85	01010101
V	86	01010110
W	87	01010111
X	88	01011000
Y	89	01011001
Z	90	01011010
[	91	01011011
\	92	01011100
]	93	01011101
^	94	01011110
_	95	01011111
`	96	01100000
a	97	01100001
b	98	01100010
c	99	01100011
d	100	01100100
e	101	01100101
f	102	01100110
g	103	01100111
h	104	01101000
i	105	01101001
j	106	01101010
k	107	01101011
l	108	01101100
m	109	01101101
n	110	01101110
o	111	01101111
p	112	01110000
q	113	01110001
r	114	01110010
s	115	01110011
t	116	01110100
u	117	01110101
v	118	01110110

(continues)

(continued)

Character	ASCII	Binary	Character	ASCII	Binary	Character	ASCII	Binary
w	119	01110111	|	124	01111100			
x	120	01111000	}	125	01111101			
y	121	01111001	~	126	01111110			
z	122	01111010	DELETE	127	01111111			
{	123	01111011						

APPENDIX D

How to Use Microsoft Visual C++

This appendix is available online at the Course Technology Web site. You can obtain the appendix by visiting *www.cengage.com/coursetechnology* and then navigating to the page for this book.

APPENDIX E

How to Use Dev-C++

This appendix is available online at the Course Technology Web site. You can obtain the appendix by visiting *www.cengage.com/coursetechnology* and then navigating to the page for this book.

APPENDIX F

Classes and Objects

After studying Appendix F, you should be able to:

- Differentiate between procedure-oriented and object-oriented programming
- Define the terms used in object-oriented programming
- Create a class definition
- Instantiate an object from a class that you define
- Create a default constructor
- Create a parameterized constructor
- Include methods other than constructors in a class
- Overload the methods in a class

Object-Oriented Terminology

In Chapter 1, you learned that some programs are procedure oriented and some are object oriented. All of the programs you created in the previous chapters were procedure oriented. Recall that, when writing a procedure-oriented program, the programmer concentrates on the major tasks that the program must perform to accomplish its goal. A payroll program, for example, typically performs several major tasks, such as inputting the employee data, calculating the gross pay, calculating the taxes, calculating the net pay, and outputting a paycheck. The programmer usually assigns each major task to a function, which is the primary component in a procedure-oriented program. The primary component in an object-oriented program, on the other hand, is an object. An **object** is anything that can be seen, touched, or used. In other words, an object is nearly any *thing.* When writing an object-oriented program, the programmer focuses on the objects (rather than the tasks) that the program can use to accomplish its goal. The objects can take on many different forms. Programs written for the Windows environment typically use objects such as check boxes, list boxes, and buttons. A payroll program, on the other hand, might utilize objects found in real life, such as a time card object, an employee object, or a paycheck object. Because each object is viewed as an independent unit, an object can be used in more than one program, usually with little or no modification. A check object used in a payroll program, for example, also can be used in a sales revenue program (which receives checks from customers) and an accounts payable program (which issues checks to creditors). The ability to use an object for more than one purpose enables code-reuse, which saves programming time and money—advantages that contribute to the popularity of object-oriented programming.

Every object in an object-oriented program is created from a **class**, which is a pattern or blueprint that the computer uses when creating the object. Using object-oriented programming (**OOP**) terminology, objects are **instantiated** (created) from a class, and each object is referred to as an **instance** of the class. A `string` object (variable or named constant), for example, is an instance of the `string` class. The input and output file objects discussed in Chapter 14 are instances of the `ifstream` and `ofstream` classes, respectively. Keep in mind that the class itself is not an object; only an instance of a class is an object. Every object has attributes and behaviors. The **attributes** are the characteristics that describe the object. When you tell someone that your wristwatch is a Valenti Model VI, you are describing the watch (an object) in terms of some of its attributes—in this case, its maker and model number. A watch also has many other attributes, such as a crown, dial, hour hand, minute hand, and movement. An object's **behaviors** fall into two categories: actions that the object is capable of performing and actions to which the object can respond. A watch, for example, can keep track of the time and date. Some watches also can illuminate their dials when a button on the watch is pushed. All of an object's attributes and behaviors are contained—or, in OOP terms, **encapsulated**—in the class from which the object is instantiated.



The term "encapsulate" means "to enclose in a capsule." In the context of OOP, the "capsule" is a class.

"Abstraction" is another term used in OOP discussions. **Abstraction** refers to the hiding of the internal details of an object from the user. Hiding the internal details helps prevent the user from making inadvertent changes to the object. The internal mechanism of a watch, for example, is enclosed (**hidden**)

in a case to protect the mechanism from damage. Attributes and behaviors that are not hidden are said to be **exposed** to the user. Exposed on a Valenti Model VI watch are the crown used to set the hour and minute hands and the button used to illuminate the dial. The idea behind abstraction is to expose to the user only those attributes and behaviors that are necessary to use the object and to hide everything else.

Another OOP term, **inheritance**, refers to the fact that you can create one class from another class. The new class, called the **derived class**, inherits the attributes and behaviors of the original class, called the **base class**. For example, the Valenti company might create a blueprint of the Model VII watch from the blueprint of the Model VI watch. The Model VII blueprint (the derived class) will inherit all of the attributes and behaviors of the Model VI blueprint (the base class), but it then can be modified to include an additional feature, such as an alarm.

Finally, you also will hear the term "polymorphism" in OOP discussions. **Polymorphism** is the object-oriented feature that allows the same instruction to be carried out differently depending on the object. For example, you open a door, but you also open an envelope, a jar, and your eyes. Similarly, you can set the time, date, and alarm on a Valenti watch. Although the meaning of the verbs "open" and "set" are different in each case, you can understand each instruction because the combination of the verb and the object makes the instruction clear.



You can use the acronym APIE (Abstraction, Polymorphism, Inheritance, and Encapsulation) to help you remember some of the OOP terms.



The answers to Mini-Quiz questions are located in the Cpp6\AppF\AppendixF.pdf file.

Mini-Quiz F-1

1. OOP is an acronym for _____.
2. A class is an object.
   a. True
   b. False
3. An object created from a class is called _____.
   a. an attribute
   b. an instance of the class
   c. the base class
   d. the derived class
4. The actions that an object can perform or to which an object can respond are called the object's _____.

Defining a Class in C++

In previous chapters, you instantiated objects using existing classes, such as the `string` and `ofstream` classes. You used the instantiated objects in a variety of ways in many different programs. In some programs, you used

a `string` object (variable) to store a name, whereas in others you used it to store a phone number. Similarly, one of the programs in Chapter 14 used an output file object to save CD (compact disc) information. Another program in the same chapter used an output file object to save a store's sales information. You also can define your own classes and then create instances (objects) from those classes. As do the `string` and `ofstream` classes, your classes must specify the attributes and behaviors of the objects they create. You specify the attributes and behaviors using a **class definition**. Figure F-1 shows the syntax used in this book to define a class. The figure also includes an example of defining a class name `FormattedDate`. Notice that the syntax contains two sections: a declaration section and an optional implementation section. The **declaration section** contains the C++ **class statement**, which begins with the keyword `class` followed by the name of the class; the statement ends with a semicolon. Although it is not a requirement, the convention is to enter the class name using **Pascal case**, which means you capitalize the first letter in the name and the first letter in any subsequent words in the name. Examples of class names that follow this naming convention include `Check`, `FormattedDate`, and `TimeCard`. Within the `class` statement, you list the attributes and behaviors of the objects that the class will create and you enclose the attributes and behaviors in a set of braces. In most cases, the attributes (called data members) are represented by variable declarations, and the behaviors (called member methods) are represented by method prototypes. A **method** is simply a function that is defined in a class definition. You enter the method definitions in the **implementation section** of a class definition. The implementation section will contain one definition for each prototype listed in the declaration section. If no method prototypes appear in the declaration section, the implementation section is not needed.



Although you can code a class in C++ in a matter of minutes, the objects produced by such a class probably will not be of much use. The creation of a good class, which is one whose objects can be used in a variety of ways by many different programs, requires a lot of time, patience, and planning.



Many C++ programmers refer to the methods in a class as member functions.

HOW TO Define a Class

Syntax

```
//declaration section
class className
{
public:
    public attributes (data members)
    public behaviors (member methods)
private:
    private attributes (data members)
    private behaviors (member methods)
};

[//implementation section
member method definitions]
```

colon

colon

semicolon

(continues)

Figure F-1 How to define a class

(continued)

Example

```
//declaration section
class FormattedDate
{
public:
     FormattedDate();
     void setDate(string, string, string);
     string getFormattedDate();
private:
     string month;
     string day;
     string year;
};

//implementation section
FormattedDate::FormattedDate()
{
     //initializes the private variables
     month = "0";
     day   = "0";
     year  = "0";
}  //end of default constructor

void FormattedDate::setDate(string m , string d, string y)
{
     //assigns program values to the private variables
     month = m;
     day   = d;
     year  = y;
}  //end of setDate method

string FormattedDate::getFormattedDate()
{
     //formats and returns values stored in the private
     //variables
     return month + "/" + day + "/" + year;
}  //end of getFormattedDate method
```

method prototypes

variable declarations

Figure F-1 How to define a class

As Figure F-1 shows, a class can contain both public members and private members. You record the public members below the keyword `public` in the `class` statement. The private members are recorded below the keyword `private`. When you use a class to instantiate (create) an object in a program, only the public members of the class are exposed (made available) to the program; the private members are hidden. In most cases, you will want to expose the member methods and hide the data members. Therefore, in most class definitions, you will list the method prototypes below the keyword

`public` in the `class` statement and list the variable declarations below the keyword `private`, as shown in the `FormattedDate` class definition in Figure F-1. You expose the member methods to allow the program to use the service each method provides. You hide the variables (data members) to protect their contents from being changed inadvertently by the program. When a program needs to assign data to a private variable, it must use a public member method to do so. For example, when using the `FormattedDate` class in Figure F-1, the program would need to use the `setDate` method to assign data to the `month`, `day`, and `year` variables. It is the public member method's responsibility to validate the data, if necessary, and then either assign the data to the private data member (if the data is valid) or reject the data (if the data is not valid). Keep in mind that a program does not have direct access to the private members of a class. Rather, it must access the private members indirectly, through a public member method.

Instantiating an Object and Referring to a Public Member

Figure F-2 shows the syntax for instantiating an object in a C++ program. In the syntax, *className* and *objectName* are the names of the class and object, respectively. The figure also includes an example of instantiating a `FormattedDate` object named `reportDate`.

HOW TO Instantiate an Object

Syntax

className objectName; — semicolon

Example

```
FormattedDate reportDate;
```

uses the `FormattedDate` class from Figure F-1 to instantiate an object named `reportDate`

Figure F-2 How to instantiate an object

After an object has been instantiated in a program, the program can refer to a public member of the class using the syntax shown in Figure F-3. In the syntax, *objectName* and *publicMember* are the names of the object and public member, respectively. The figure also includes examples of referring to the `reportDate` object's `getFormattedDate` and `setDate` methods. Both methods are public members of the `FormattedDate` class from which the `reportDate` object was instantiated.

HOW TO Refer to a Public Member of an Object's Class

Syntax
objectName.publicMember; — semicolon

Example 1
`reportDate.getFormattedDate();`
refers to the `reportDate` object's `getFormattedDate` method, which is a public method of the `FormattedDate` class (shown earlier in Figure F-1)

Example 2
`reportDate.setDate(monthNum, dayNum, yearNum);`
refers to the `reportDate` object's `setDate` method, which is a public method of the `FormattedDate` class (shown earlier in Figure F-1)

Figure F-3 How to refer to a public member of an object's class

Mini-Quiz F-2



The answers to Mini-Quiz questions are located in the Cpp6\AppF\AppendixF.pdf file.

1. A program cannot access a public member method directly.
   a. True
   b. False

2. In C++, you enter the `class` statement in the _____ section of a class definition, and you enter the method definitions in the _____ section.

3. Typically, the data members (attributes) of a class are represented by _____ in a class definition.
   a. constant declarations
   b. method prototypes
   c. method definitions
   d. variable declarations

4. A private data member can be accessed directly by a public member method.
   a. True
   b. False

5. Write the C++ statement to instantiate a `Check` object named `payCheck`.

6. How do you refer to the `payCheck` object's `getCheck` method?
   a. `payCheck.getCheck()`
   b. `payCheck::getCheck()`

c. `getCheck()`

d. `getCheck().payCheck`

Example 1—A Class that Contains Public Data Members Only

Sweets Unlimited employs several salespeople. The sales manager wants a program that allows him to enter a salesperson's name and sales amount. The program should calculate the salesperson's 5% bonus and then save the salesperson's name, sales amount, and bonus amount in a sequential access file named salesInfo.txt. In the context of object-oriented programming (OOP), a salesperson can be treated as an object having three attributes: a name, a sales amount, and a bonus amount. By including the attributes in a class, you can create a pattern that a program can use to instantiate a salesperson object. In this case, you will enter the three attributes of a salesperson in a class named `Salesperson`, which the Sweets Unlimited program will use to instantiate a `Salesperson` object named `employee`. The program will use the `employee` object to store a salesperson's information before the information is written to the sequential access file. Figure F-4 shows the Sweets Unlimited program. The code pertaining to the `Salesperson` class and `employee` object is shaded in the figure.

```
 1  //Sweets Unlimited.cpp
 2  //Saves data to a sequential access file
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  #include <fstream>
 7  #include <string>
 8  using namespace std;
 9
10  //declaration section
11  class Salesperson
12  {
13  public:
14        string name;
15        double sales;
16        double bonus;
17  };
18
19  int main()
20  {
21     //declare constant and file object
22     const double BONUS_RATE = .05;
23     ofstream outFile;
24
25     //create Salesperson object
26     Salesperson employee;
```

Figure F-4 Sweets Unlimited program *(continues)*

(continued)

```
27
28     //open sequential access file
29     outFile.open("salesInfo.txt", ios::app);
30
31     if (outFile.is_open())
32     {
33        cout << "Salesperson's name (X to exit): ";
34        getline(cin, employee.name);
35        while (employee.name != "X"
36             && employee.name != "x")
37        {
38             cout << "Sales amount: ";
39             cin >> employee.sales;
40             cin.ignore(100, '\n');
41
42             //calculate the bonus
43             employee.bonus =
44                  employee.sales * BONUS_RATE;
45
46             //write the salesperson's information
47             //to a sequential access file
48             outFile << employee.name << "#"
49                  << employee.sales << "#"
50                  << employee.bonus << endl;
51
52             cout << "Salesperson's name (X to exit): ";
53             getline(cin, employee.name);
54        }  //end while
55     }
56     else
57        cout << "The file could not be opened." << endl;
58     //end if
59
60     system("pause");
61     return 0;
62  }  //end of main function
```

Figure F-4 Sweets Unlimited program

The definition of the `Salesperson` class appears in Lines 11 through 17 in Figure F-4. The class contains three data members only. Each data member is represented by a variable declaration, which specifies the variable's data type and name but not its initial value. This is because you cannot initialize variables within the `class` statement. Notice that the variable declarations in Lines 14 through 16 appear below the keyword `public`. When a variable is declared below the `public` keyword in a class definition, any program that contains an instance of the class can access the variable. In this case, any program that instantiates a `Salesperson` object can use the variables included in the `Salesperson` class. Notice that the `Salesperson` class definition contains the declaration section only. The implementation section is not necessary in this class definition, because no method prototypes appear in the declaration section. (Recall that the implementation section contains the definitions of the methods whose prototypes are listed in the declaration

section.) The `Salesperson employee;` statement in Line 26 uses the `Salesperson` class definition to instantiate a `Salesperson` object named `employee`. The `employee` object contains three data members: a `string` variable named `name` and two `double` variables named `sales` and `bonus`. As the program in Figure F-4 shows, you use `employee.name` to refer to the `name` variable in the `employee` object. You use `employee.sales` and `employee.bonus` to refer to the `employee` object's `sales` and `bonus` variables. Figure F-5 shows a sample run of the Sweets Unlimited program, and Figure F-6 shows the contents of the salesInfo.txt file created by the program.

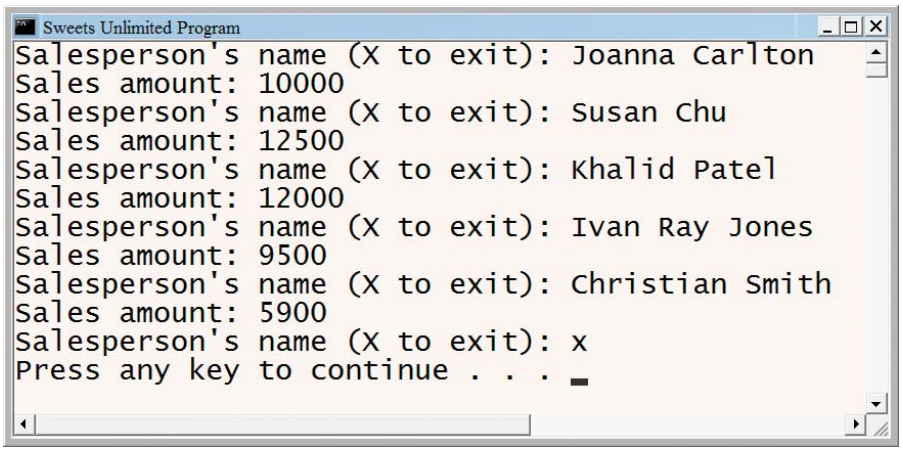

Sweets Unlimited Program
Salesperson's name (X to exit): Joanna Carlton
Sales amount: 10000
Salesperson's name (X to exit): Susan Chu
Sales amount: 12500
Salesperson's name (X to exit): Khalid Patel
Sales amount: 12000
Salesperson's name (X to exit): Ivan Ray Jones
Sales amount: 9500
Salesperson's name (X to exit): Christian Smith
Sales amount: 5900
Salesperson's name (X to exit): x
Press any key to continue . . .

Figure F-5 Sample run of the Sweets Unlimited program

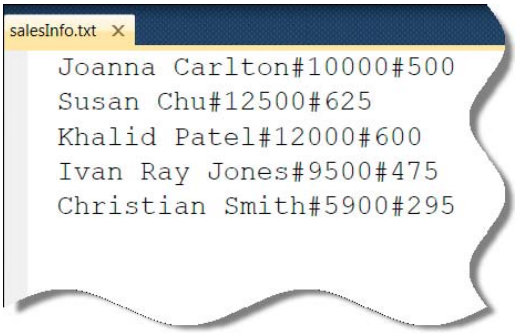


Figure F-6 Contents of the salesInfo.txt file displayed in a text editor

Header Files

Although you can enter a class definition in the program that uses the class, as shown earlier in Figure F-4, most programmers enter a class definition in a separate file called a **header file**. Figure F-7 shows the definition of the `Salesperson` class entered in a header file named Salesperson.h. Unlike program filenames, which end with .cpp, header filenames end with .h. (You will learn how to add a header file to a solution in Lab F-2, which is contained in the Cpp6\AppF\AppendixF.pdf file.) Figure F-8 shows a modified version of the Sweets Unlimited program. Unlike the original program, the modified program does not contain the Salesperson class definition. Instead, the modified program uses the class definition contained in the Salesperson.h header file. Typically, a header file is stored in the same location as the program file

that employs the class. In this case, for example, the Salesperson.h file would be stored in the same location as the Modified Sweets Unlimited.cpp file. The programmer uses a `#include` directive to tell the compiler to include the contents of the header file in the program. In the modified Sweets Unlimited program, the `#include "Salesperson.h"` directive (which is shaded in Figure F-8) tells the compiler to merge the contents of the Salesperson.h file with the contents of the current program. In other words, it tells the compiler to include the `Salesperson` class definition in the current program. Notice that the header filename is enclosed in quotation marks. The quotation marks indicate that the header file is located in the same folder as the program file.



The angle brackets (<>) in the other directives in Figure F-8 indicate that those files are located in the folder that contains the C++ Standard Library header files.

```
 1  //Salesperson.h
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <string>
 5  using namespace std;
 6
 7  //declaration section
 8  class Salesperson
 9  {
10  public:
11       string name;
12       double sales;
13       double bonus;
14  };
```

Figure F-7 `Salesperson` class definition entered in the Salesperson.h header file

```
 1  //Modified Sweets Unlimited.cpp
 2  //Saves data to a sequential access file
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  #include <fstream>
 7  #include <string>
 8  #include "Salesperson.h"
 9  using namespace std;
10
11  int main()
12  {
13     //declare constant and file object
14     const double BONUS_RATE = .05;
15     ofstream outFile;
16
17     //create Salesperson object
18     Salesperson employee;
```

Figure F-8 Partial C++ code for the modified Sweets Unlimited program

Although you can define a class that contains only public variables, like the `Salesperson` class shown in Figure F-7, it is rarely done. The disadvantage of using public variables in a class is that a class cannot control the values

assigned to its public variables. Therefore, it cannot validate the values to ensure that they are appropriate for the variables. Besides, most classes contain not only variables, but methods as well. This is because the purpose of a class in OOP is to encapsulate the attributes (variables) that describe an object and the behaviors (methods) that allow the object to perform tasks. The class used in the next example will show you how to include data validation and methods in a class.

Example 2—A Class that Contains a Private Data Member and Public Member Methods

In this example, you will view the code for a class named `Square`. A `Square` object has one attribute: the length of one of its sides. It also has four behaviors: it can initialize its side measurement when it is created; it can assign a value to its side measurement after it has been created; it can provide its side measurement value; and it can calculate and return its area. Figure F-9 shows the `Square` class defined in the Square.h header file. The `Square` class contains one private data member: an `int` variable named `side`. When a variable is declared below the `private` keyword in a `class` statement, it can be used only by the code entered in the class definition. In this case, the code uses the `side` variable to store the side measurement of a `Square` object. The `Square` class also contains four public member methods named `Square`, `setSide`, `getSide`, and `calculateArea`. The method prototypes for these methods appear below the `public` keyword in the `class` statement. The definitions of the methods appear in the implementation section of the class definition.

```
//Square.h
//Created/revised by <your name> on <current date>

//declaration section
class Square
{
public:
      Square();
      void setSide(int);
      int getSide();
      int calculateArea();
private:
      int side;
};

//implementation section
Square::Square()
{
      side = 0;
}  //end of default constructor
```

Figure F-9 `Square` class definition entered in the Square.h header file *(continues)*

(continued)

```
void Square::setSide(int sideValue)
{
      if (sideValue > 0)
            side = sideValue;
      else
            side = 0;
      //end if
}  //end of setSide method

int Square::getSide()
{
      return side;
}  //end of getSide method

int Square::calculateArea()
{
      return side * side;
}  //end of calculateArea method
```

Figure F-9 `Square` class definition entered in the Square.h header file

In the `Square` class definition in Figure F-9, both the first method prototype and the first method definition pertain to the default constructor. A **constructor** is a method whose instructions the computer automatically processes each time an object is instantiated from the class. The sole purpose of a constructor is to initialize the class's private variables. Every class should have at least one constructor. Each of a class's constructors must have the same name as the class, but its formal parameters (if any) must be different from any other constructor in the class. A constructor that has no formal parameters is called the **default constructor**. A class can have only one default constructor. Because a constructor does not return a value, its prototype and definition do not begin with a data type. However, notice that its definition begins with the name of the class followed by the scope resolution operator (`::`), the name of the constructor, and a set of empty parentheses—in this case, `Square::Square()`. The scope resolution operator indicates that the `Square` method is a member of (or is contained in) the `Square` class. The `Square` method's definition in Figure F-9 contains the code to initialize the `Square` class's private `side` variable to the number 0.

As you learned earlier, a program does not have direct access to a private variable in a class. Rather, it must access the private variable indirectly, through a public method. A program that instantiates a `Square` object can use the public `setSide` method to assign a value to the private `side` variable. In this case, the `setSide` method receives the value from the program that invokes it and then stores the value in its formal parameter: an `int` variable named `sideValue`. The code contained in the method definition verifies that the value received from the program is greater than zero. If it is, the code assigns the value to the private `side` variable; otherwise, it assigns the number 0 to the variable. Notice that the `setSide` method's prototype and definition begin with the keyword `void`, which indicates that the method does not return a value. A program that instantiates a `Square` object can use the public `getSide` method, on the other hand, to retrieve the

value stored in the private `side` variable. Unlike the `setSide` method, the `getSide` method is a value-returning method. As Figure F-9 indicates, the `getSide` method returns an integer. The last method in the `Square` class, `calculateArea`, is also a value-returning method. In this case, it returns an integer that represents the area of a `Square` object. The method calculates the area by multiplying the private `side` variable's value by itself. Figure F-10 shows the area calculator program, which uses a `Square` object. The code pertaining to the `Square` object is shaded in the figure. The `#include "Square.h"` directive in Line 6 tells the compiler to include the contents of the Square.h file (shown earlier in Figure F-9) in the program. The `Square squareFigure;` statement in Line 12 instantiates a `Square` object named `squareFigure`. When the object is created, the default constructor is called, automatically, to initialize the private data member. In this case, the default constructor initializes the `side` variable to the number 0. The `squareFigure.setSide(sideMeasurement);` statement in Line 20 calls the `Square` object's `setSide` method, passing it the side measurement value entered by the user. Recall that the `setSide` method is a public member of the `Square` class. The `setSide` method verifies that the value passed to it is greater than zero. If it is, the method assigns the value to the `Square` object's private `side` variable; otherwise, it assigns the number 0 to the variable. The `area = squareFigure.calculateArea();` statement in Line 22 calls the `Square` object's `calculateArea` method to calculate and return the `Square` object's area. The statement assigns the method's return value to the program's `area` variable. The `squareFigure.getSide()` code in Line 25 calls the `Square` object's `getSide` method, which simply retrieves the value stored in the private `side` variable. A sample run of the area calculator program is shown in Figure F-11.

```
 1  //Area Calculator.cpp
 2  //Displays the area of a rectangle
 3  //Created/revised by <your name> on <current date>
 4
 5  #include <iostream>
 6  #include "Square.h"
 7  using namespace std;
 8
 9  int main()
10  {
11     //create Square object
12     Square squareFigure;
13     //declare variables
14     int sideMeasurement = 0;
15     int area            = 0;
16
17     cout << "Side measurement (feet): ";
18     cin >> sideMeasurement;
19     //assign side measurement to Square object
20     squareFigure.setSide(sideMeasurement);
```

Figure F-10 Area calculator program *(continues)*

(continued)

```
21
22      area = squareFigure.calculateArea();
23      cout <<
24         "The area of a square with a side measurement of "
25         << squareFigure.getSide()
26         << " feet is " << area << " square feet." << endl;
27      system("pause");
28      return 0;
29   }  //end of main function
```

Figure F-10 Area calculator program

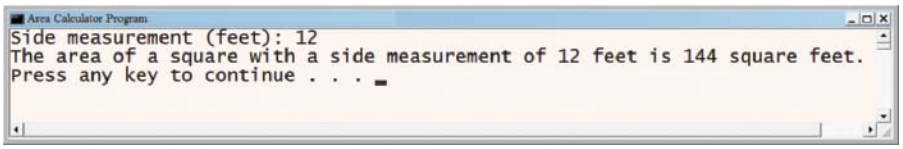

Area Calculator Program
Side measurement (feet): 12
The area of a square with a side measurement of 12 feet is 144 square feet.
Press any key to continue . . .

Figure F-11 Sample run of the area calculator program

Mini-Quiz F-3

1. The `::` operator is called the _____.
2. Write the default constructor's prototype for a class named `Item`.
3. The `Item` class in Question 2 contains two private data members: a `char` variable named `code` and an `int` variable named `price`. Write the definition for the default constructor.



The answers to Mini-Quiz questions are located in the Cpp6\AppF\AppendixF.pdf file.

Example 3—Using a Class that Contains Two Constructors

In this example, you view the code for a class named `MonthDay`. A `MonthDay` object has two attributes: a month number and a day number. A `MonthDay` object also has three behaviors. First, it can initialize its attributes using values provided by the class. Second, it can initialize its attributes using values provided by the program in which it is instantiated. Third, it can return its month number and day number attributes separated by a slash. Figure F-12 shows the `MonthDay` class defined in the MonthDay.h header file. The `MonthDay` class contains two private data members: a `string` variable named `month` and a `string` variable named `day`. It also contains four public member methods: two are named `MonthDay`, one is named `setMonthDay`, and one is named `getMonthDay`. The two `MonthDay` methods are the class's constructors. The first constructor is the default constructor, because it does not have any formal parameters. The computer invokes the default constructor when you use a statement such as `MonthDay myDate;` to instantiate a `MonthDay` object. The code contained in the default constructor initializes the class's private `month` and `day` variables to the empty string. The second constructor in

the class allows you to specify the initial values for a `MonthDay` object when the object is created. In this case, the initial values must be strings, because the constructor's *parameterList* contains two `string` variables. Constructors that contain parameters are called **parameterized constructors**. The method name combined with its optional *parameterList* is called the method's **signature**. You include the initial values, enclosed in a set of parentheses, in the statement that instantiates the object. For example, the `MonthDay myDate("04", "28");` statement instantiates a `MonthDay` object, and it passes two `string` arguments to the parameterized `MonthDay` constructor. When you instantiate an object, the computer determines which class constructor to use by matching the quantity, data type, and position of the arguments with the quantity, data type, and position of the parameters listed in each constructor's *parameterList*. In this case, the computer will invoke the default constructor when you use the `MonthDay myDate;` statement to instantiate a `MonthDay` object. However, it will use the parameterized constructor when you use the `MonthDay myDate("04", "28");` statement.

```
 1  //MonthDay.h
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <string>
 5  using namespace std;
 6
 7  //declaration section
 8  class MonthDay
 9  {
10  public:
11     MonthDay();
12     MonthDay(string, string);
13     void setMonthDay(string, string);
14     string getMonthDay();
15  private:
16     string month;
17     string day;
18  };
19
20  //implementation section
21  MonthDay::MonthDay()
22  {
23     month = "";
24     day   = "";
25  }  //end of default constructor
26
27  MonthDay::MonthDay(string x, string y)
28  {
29     setMonthDay(x, y);
30  }  //end of constructor
31
32  void MonthDay::setMonthDay(string m, string d)
33  {
34     month = m;
35     day   = d;
36  }  //end of setMonthDay method
37
```

Figure F-12 `MonthDay` class definition entered in the MonthDay.h header file *(continues)*

(continued)

```
38  string MonthDay::getMonthDay()
39  {
40     return month + "/" + day;
41  }  //end of getMonthDay method
```

Figure F-12 MonthDay class definition entered in the MonthDay.h header file

Figure F-13 shows a program that uses a MonthDay object to display a date. The code pertaining to the MonthDay object is shaded in the figure. The #include "MonthDay.h" directive in Line 6 tells the compiler to include the contents of the MonthDay.h file (shown in Figure F-12) in the program. The MonthDay myDate("04", "28"); statement in Line 12 instantiates a MonthDay object named myDate. When the object is instantiated, the parameterized constructor is called, automatically, to initialize the private data members. In this case, the parameterized constructor initializes the month and day variables to the strings "04" and "28", respectively. The cout statement in Lines 18 and 19 calls the MonthDay object's getMonthDay method. The method retrieves the values stored in the month and day variables and then returns the values (separated by a slash) to the cout statement. The statement displays the return value—in this case, 04/28—on the computer screen. The myDate.setMonthDay(myMonth, myDay); statement in Line 27 calls the setMonthDay method, passing it the values entered by the user. The method assigns the values to the private month and day variables. The cout statement in Lines 30 and 31 calls the getMonthDay method again. As before, the method retrieves the values stored in the month and day variables and then returns the values (separated by a slash) to the statement. The statement displays the return value on the computer screen. A sample run of the display date program is shown in Figure F-14.

```
 1  //Display Date.cpp - displays a date
 2  //Created/revised by <your name> on <current date>
 3
 4  #include <iostream>
 5  #include <string>
 6  #include "MonthDay.h"
 7  using namespace std;
 8
 9  int main()
10  {
11     //create MonthDay object
12     MonthDay myDate("04", "28");
13     //declare variables
14     string myMonth = "0";
15     string myDay   = "0";
16
17     //display initial date
18     cout << "Initial date: "
19        << myDate.getMonthDay() << endl;
20
```

Figure F-13 Display date program *(continues)*

(continued)

```
21  cout << endl << "Change month number to: ";
22  getline(cin, myMonth);
23  cout << "Change day number to: ";
24  getline(cin, myDay);
25
26  //assign values to MonthDay object
27  myDate.setMonthDay(myMonth, myDay);
28
29  //display new date
30  cout << "New date: "
31       << myDate.getMonthDay() << endl;
32
33     system("pause");
34     return 0;
35  }  //end of main function
```

Figure F-13 Display date program

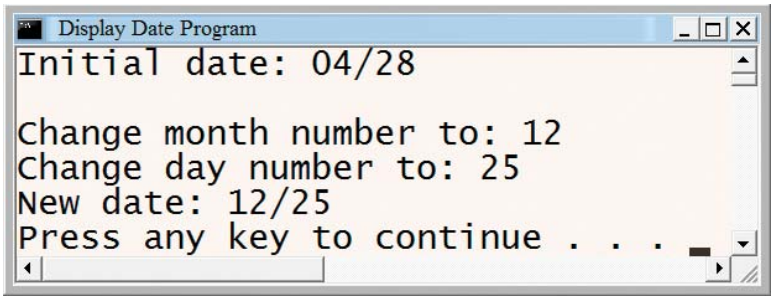

Display Date Program
Initial date: 04/28
Change month number to: 12
Change day number to: 25
New date: 12/25
Press any key to continue . . .

Figure F-14 Sample run of the display date program

Example 4—A Class that Contains Overloaded Methods

Overloaded methods are an example of polymorphism.

In this example, you view the code for a class named `GrossPay`. A `GrossPay` object has two behaviors: it can calculate and return the gross pay for a salaried employee, and it can calculate and return the gross pay for an hourly employee. The gross pay for a salaried employee is calculated by dividing the employee's annual salary by 24, because salaried employees are paid twice per month. The gross pay for an hourly employee is calculated by multiplying the number of hours the employee worked during the week by his or her pay rate. Figure F-15 shows the `GrossPay` class defined in the GrossPay.h header file. The `GrossPay` class contains two public member methods; both are named `calcGross`. Although both methods have the same name, notice that their *parameterLists* differ. The *parameterList* in the first `calcGross` method contains one formal parameter, whereas the *parameterList* in the second `calcGross` method contains two formal parameters. When two or more methods have the same name but different *parameterLists*, the methods are referred to as **overloaded methods**. Overloading is useful when two or more methods require different parameters to perform essentially the same task. Both overloaded methods in the `GrossPay` class, for example, calculate and return a gross pay amount. However, the first `calcGross` method, which

calculates and returns the gross pay amount for a salaried employee, requires a program to pass it one item of information: the employee's annual salary. The second `calcGross` method, on the other hand, calculates and returns the gross pay amount for an hourly employee and requires two items of information: the number of hours the employee worked and his or her rate of pay. Rather than using two overloaded `calcGross` methods in the `GrossPay` class, you could use two methods having different names. For example, you could use a method named `calcSalariedGross` to calculate and return the gross pay amount for a salaried employee and then use a method named `calcHourlyGross` to calculate and return the gross pay amount for an hourly employee. The advantage of overloading the `calcGross` method is that you need to remember the name of only one method.



The two `MonthDay` constructors shown earlier in Figure F-12 are overloaded methods, because both have the same name but a different *parameterList*.

```
 1  //GrossPay.h
 2  //Created/revised by <your name> on <current date>
 3
 4  //declaration section
 5  class GrossPay
 6  {
 7  public:
 8     double calcGross(double);
 9     double calcGross(double, double);
10  };
11
12  //implementation section
13  double GrossPay::calcGross(double yearSalary)
14  {
15     return yearSalary / 24;
16  }  //end of calcGross method
17
18  double GrossPay::calcGross(double h, double r)
19  {
20     return h * r;
21  }  //end of calcGross method
```

Figure F-15 `GrossPay` class definition entered in the GrossPay.h header file

Figure F-16 shows the gross pay program, which uses a `GrossPay` object to calculate and return an employee's gross pay amount. The code pertaining to the `GrossPay` object is shaded in the figure. The `#include "GrossPay.h"` directive in Line 8 tells the compiler to include the contents of the GrossPay.h file (shown in Figure F-15) in the program. The `GrossPay employGross;` statement in Line 14 instantiates a `GrossPay` object named `employGross`. Notice that the `calcGross` method appears in two statements in Figure F-16. The computer uses the signature of the `calcGross` method in each statement to determine which of the class's `calcGross` methods to process. In this case, the `gross = employGross.calcGross(salary);` statement in Line 32 tells the computer to process the `calcGross` method that contains one `double` parameter and then assign the return value to the program's `gross` variable. In the `GrossPay` class, the `calcGross` method that contains one `double` parameter calculates and returns the gross pay amount for a salaried worker. The `gross = employGross.calcGross(hours, hrlyPay);` statement in Lines 41 and 42, on the other hand, tells the computer to process the

calcGross method that contains two double parameters and then assign the return value to the program's gross variable. In the GrossPay class, the calcGross method that contains two double parameters calculates and returns the gross pay amount for an hourly worker. Before the program ends, it displays the contents of its gross variable on the computer screen. Sample runs of the gross pay program are shown in Figures F-17 and F-18.

```
 1  //Gross Pay.cpp
 2  //Displays the gross pay for salaried and
 3  //hourly employees
 4  //Created/revised by <your name> on <current date>
 5
 6  #include <iostream>
 7  #include <iomanip>
 8  #include "GrossPay.h"
 9  using namespace std;
10
11  int main()
12  {
13     //create GrossPay object
14     GrossPay employGross;
15     //declare variables
16     int employType = 0;
17     double salary  = 0.0;
18     double hours   = 0.0;
19     double hrlyPay = 0.0;
20     double gross   = 0.0;
21
22     cout << fixed << setprecision(2);
23
24     //determine employee's type
25     cout << "Salaried (1) or Hourly (2): ";
26     cin >> employType;
27
28     if (employType == 1)
29     {
30        cout << "Annual salary: ";
31        cin >> salary;
32        gross = employGross.calcGross(salary);
33     }
34     else
35        if (employType == 2)
36        {
37           cout << "Hours worked: ";
38           cin >> hours;
39           cout << "Hourly pay: ";
40           cin >> hrlyPay;
41           gross =
42              employGross.calcGross(hours, hrlyPay);
43        }
44        else
45           cout << "Incorrect employee type."
46           c<< endl;
47        //end if
48     //end if
49
```

Figure F-16 Gross pay program *(continues)*

(continued)

```
50     //display gross pay
51     cout << "Gross pay: $" << gross << endl;
52
53     system("pause");
54     return 0;
55  }  //end of main function
```

Figure F-16 Gross pay program

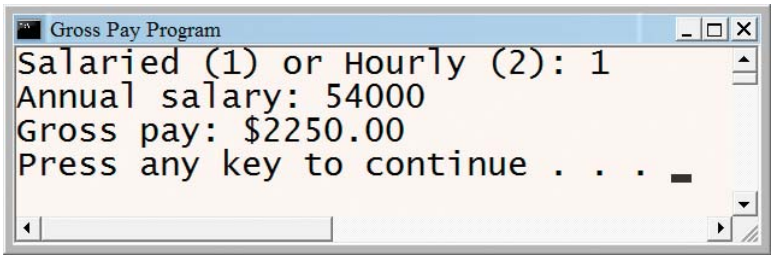


Figure F-17 Sample run of the gross pay program

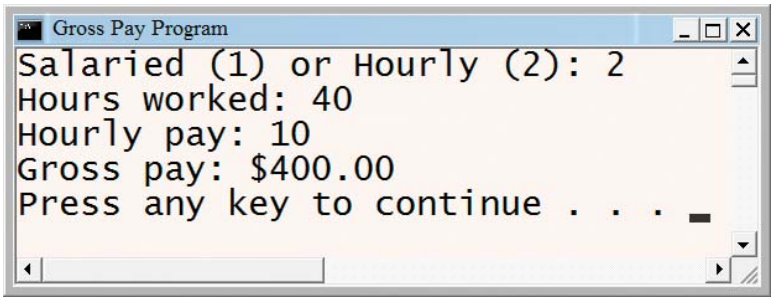


Figure F-18 Another sample run of the gross pay program

Mini-Quiz F-4



The answers to Mini-Quiz questions are located in the Cpp6\AppF\AppendixF.pdf file.

1. A method's name along with its optional *parameterList* is called the method's _____.

2. Write the prototype for a parameterized constructor in the `Item` class. The constructor has one formal parameter, which has the `int` data type.

3. If a class contains two methods that have the same name but different *parameterLists*, the methods are referred to as _____ methods.



LABS F-1 Through F-5

The labs are located in the AppendixF.pdf file, which is contained in the Cpp6\AppF folder. The file also contains the answers to the labs.

Summary

- A class is a pattern for creating one or more instances of the class. Each instance is considered an object.
- A class encapsulates all of an object's attributes and behaviors. An object's attributes are the characteristics that describe the object. Its behaviors are the actions that the object can perform or to which the object can respond.
- The OOP term "abstraction" refers to the hiding of an object's internal details from the user. Hiding the internal details prevents the user from making inadvertent changes to the object.
- The idea behind abstraction is to expose to the user only the attributes and behaviors that are necessary to use the object and to hide everything else. In most classes, you expose an object's behaviors (member methods) and you hide its attributes (data members).
- Polymorphism is the object-oriented feature that allows the same instruction to be carried out differently depending on the object.
- You use a class definition to create a class. The class definition contains two sections: declaration and implementation. The declaration section contains the `class` statement. The implementation section contains the method definitions.
- You instantiate (create) an object using the syntax *className objectName*;, in which *className* is the name of the class and *objectName* is the name of the object.
- You refer to a public member of a class using the syntax *objectName*.*publicMember*.
- Most C++ programmers enter class definitions in header files. Header filenames end with .h.
- You can use a constructor to initialize the data members in a class when an object is instantiated. A class can have more than one constructor, but only one can be the default constructor. The default constructor has no formal parameters.
- Each constructor in a class has the same name, but its formal parameters (if any) must be different than any other constructor in the class. A constructor that has one or more formal parameters is called a parameterized constructor.
- A constructor does not have a data type, because it cannot return a value.
- You can overload the methods in a class. Doing this allows you to use the same name for methods that require different information to perform the same task. The computer uses the method's signature to determine which overloaded method to process.

Key Terms

Abstraction—the OOP term that refers to the hiding of the internal details of an object from the user

Attributes—the characteristics that describe an object

Base class—the class from which a derived class is created

Behaviors—the actions that an object is capable of performing or to which the object can respond

Class—a pattern or blueprint used to instantiate an object in a program

Class definition—used to specify the attributes and behaviors of an object

`class` statement—the statement used to create a class in C++

Constructor—a class method whose instructions the computer automatically processes each time an object is instantiated from the class

Declaration section—the section that contains the `class` statement in a class definition

Default constructor—a constructor that has no formal parameters

Derived class—the class that inherits the attributes and behaviors of a base class

Encapsulated—the OOP term that refers to the grouping together of the attributes and behaviors of an object within a class

Exposed—the OOP term that refers to the attributes and behaviors that the user can access

Header file—a file that contains a class definition; header filenames end with .h

Hidden—the OOP term that refers to the attributes and behaviors that the user cannot access

Implementation section—the section that contains the method definitions in a class definition

Inheritance—the OOP term that refers to the fact that you can create one class (the derived class) from another class (the base class); the derived class inherits the attributes and behaviors of the base class

Instance—in OOP terminology, an object instantiated (created) from a class

Instantiated—the OOP term that refers to objects being created from a class

Method—a function that is defined in a class definition

Object—anything that can be seen, touched, or used

OOP—an acronym for object-oriented programming

Overloaded methods—two or more class methods that share the same name but have different *parameterLists*

Parameterized constructors—constructors that have one or more formal parameters

Pascal case—the practice of capitalizing the first letter in a name and the first letter in any subsequent words in the name

Polymorphism—the object-oriented feature that allows the same instruction to be carried out differently depending on the object

Signature—the combination of a method's name with its optional *parameterList*

Review Questions

1. A blueprint for creating an object in C++ is called _____.
   a. a class
   b. an instance
   c. a map
   d. a pattern

2. Which of the following statements is false?
   a. An example of an attribute is the `minutes` variable in a `Time` class.
   b. An example of a behavior is the `setTime` method in a `Time` class.
   c. An object created from a class is referred to as an instance of the class.
   d. A class is considered an object.

3. You hide a member of a class by recording the member below the _____ keyword in the `class` statement.
   a. `confidential`
   b. `hidden`
   c. `private`
   d. `restricted`

4. You expose a member of a class by recording the member below the _____ keyword in the `class` statement.
   a. `common`
   b. `exposed`
   c. `public`
   d. `unrestricted`

5. A program can access the private members of a class _____.
   a. directly
   b. only through the public members of the class
   c. only through other private members of the class
   d. none of the above, because the program cannot access the private members of a class in any way

6. In most classes, you expose the _____ and hide the _____.
   a. attributes, data members
   b. data members, member methods
   c. member methods, data members
   d. variables, member methods

7. The method definitions for a class are entered in the _____ section in the class definition.
   a. declaration
   b. implementation
   c. method
   d. program-defined

8. Which of the following is the scope resolution operator?
   a. `::` (two colons)
   b. * (asterisk)
   c. `.` (period)
   d. `->` (hyphen and a greater than symbol)

9. The name of the constructor for a class named `Animal` is _____.
   a. `Animal`
   b. `AnimalConstructor`
   c. `ConstAnimal`
   d. any of the above could be used as the name of the constructor

10. Which of the following statements is false?
    a. You typically use a public member method to change the value stored in a private data member.
    b. Because a constructor does not return a value, you place the keyword `void` before the constructor's name.
    c. The public member methods in a class can be accessed by any program that uses an object created from the class.
    d. An instance of a class is considered an object.

11. Which of the following creates an `Animal` object named `dog`?
    a. `Animal dog;`
    b. `Animal "dog";`
    c. `dog = "Animal";`
    d. `dog Animal();`

12. A program creates an `Animal` object named `dog`. Which of the following calls the `displayBreed` method, which is a public member method contained in the `Animal` class?

    a. `Animal::displayBreed();`

    b. `displayBreed();`

    c. `dog::displayBreed();`

    d. `dog.displayBreed();`

Exercises

The Exercises are located in the AppendixF.pdf file, which is contained in the Cpp6\AppF folder.

Index

Notes

- Page numbers in **bold** indicate definitions and/or descriptions.
- Page numbers followed by *(2)* indicate two separate discussions.
- Page numbers followed by *(3)* indicate three separate discussions.
- Page numbers followed by *ans* indicate answers to TRY THIS questions.
- Page numbers followed by *blb* indicate TIPS (light bulb margin notes).
- Page numbers followed by *+blb* indicate discussions plus TIPS (light bulb margin notes).
- Page numbers followed by *fig* indicate figures.
- Page numbers followed by *+fig* indicate discussions plus figures.

Symbols

& (ampersand): address-of operator, 379
&& (ampersands). *See* And operator
<> (angle brackets): #include directive delimiters, 705*blb*
* (asterisk). *See* multiplication operator
*= (asterisk–equal sign): multiplication assignment operator, 97
\ (backslash): escape character, 527
\<*character*>: escape sequence, 527
\n: newline character, 527
{} (braces). *See* braces
:: (colons): scope resolution operator, 586, 707
"" (double quotation marks). *See* double quotation marks
= (equal sign). *See* assignment operator
== (equal signs). *See* equality operator
! (exclamation point): Not operator, 589
!= (exclamation point–equal sign). *See* not equal to operator
/ (forward slash). *See* division operator
/= (forward slash–equal sign): division assignment operator, 97
// (forward slashes): comment delimiter, 95
> (greater than sign). *See* greater than operator
>= (greater than/equal sign). *See* greater than or equal to operator
>> (greater than signs). *See* extraction operator
< (less than sign). *See* less than operator
<= (less than/equal sign). *See* less than or equal to operator
<< (less than signs). *See* insertion operator
- (minus sign). *See* negation operator; subtraction operator
-= (minus–equal sign): subtraction assignment operator, 97
(number sign): field separator character, 591
#include directives, **95**
  cmath file, 285, 311, 312
  ctime file, 317
  delimiters for, 705*+blb*
  fstream file, 585
  header files, 705
  iostream file, 585
  string file, 525
() (parentheses). *See* parentheses
% (percent sign). *See* modulus operator
%= (percent–equal sign): modulus assignment operator, 97
| (pipe symbol), 133
|| (pipe symbols). *See* Or operator
+ (plus sign): concatenation operator, 558
  See also addition operator
+= (plus–equal sign): addition assignment operator, 97
; (semicolon). *See* semicolon
'' (single quotation marks): character delimiters, 61
[] (square brackets). *See* square brackets
_ (underscore): beginning memory location names with, 54*blb*

A

ABC Company program, 372–375
abstraction (in OOP), 696–697
accessing private data members, 707–708
accessing public members, 700–701
accumulating values stored in two-dimensional arrays, 498–500
accumulators, **224**–228
  initializing and updating, 224–225, 229*+blb*
actual arguments (of functions), **140**, **311**, 326, 376
  argumentList, 326, 372
  naming, 377
  passing to functions. *See* passing variables to functions
  seed argument (srand function), 317
  See also formal parameters (of functions)

addition assignment operator (+=), 97
addition operator (+), 83
  order of precedence, 83, 139
address-of operator (&), **379**–380
  as not used in a passing arrays, 433
addresses. *See* memory locations
age message program, 377–379, 379–383
algorithms, **6**
  modifying, 12, 24, 40
  See also coding algorithms; desk-checking algorithms; planning and creating algorithms
`allRecords` function, 604, 606–607
ampersand (&). *See* address-of operator
ampersands (&&). *See* And operator
analyzing flowcharts: multiple alternative selection structures, 188–189
analyzing problems, 11, 23, 25–27
  area of a circle, 66
  gas mileage, 35
  hotel guest bill, 37
  weekly pay, 36–37, 65–66
analyzing programs:
  bonus program, 173
  car payment program, 349
  convert dollars program, 390–393
  electric bill program, 393
  Falcon Incorporated program, 511
  food fat calories and percentage program, 145
  `if` statements, 144–145
  movies program, 602–603
  number guessing game, 347–348
  phone number processing program, 560–562
  product price program, 188
  Professor Chang program, 243
  rainfall program, 466
  repetition structures, 241–242
  sales commission program, 189–190
  Sweets-4-You program, 604
  total sales programs, 290–292, 465–466, 509–511
And operator (&&), 132, 134, 135, 136*fig*, 137, 138*fig*
  order of precedence, 133, 139
angle brackets (<>): #include directive delimiters, 705*blb*
annual income program, 547–548, 550–551, 552–553
answers to mini-quizzes and labs, 626–689
area calculator programs, 336–339, 708–709
area of a circle problem, 66–68
argumentList (actual arguments), 326, 372
arguments:
  `for` clause arguments, 232–233, 429–430, 431–432
  of functions. *See* actual arguments
arithmetic assignment operators, **97**
arithmetic expressions, 83
  implicit type conversion in the processing of, 84–85
arithmetic operators, 83–87
  assignment operators, **97**
  order of precedence, 83, 128, 139
arrays, **420**
  and program efficiency, 420
  See also one-dimensional arrays; two-dimensional arrays
ASCII codes, 58–60
  vs. binary number system, 59–60
  listed, 59*fig*, 691–692*fig*
assembler, **4**
assembly languages, **4**
`assign` function, 557–558, 569*fig*
assignment operator (=), 62, **87**
  arithmetic assignment operators, **97**
  vs. equality operator, 128
assignment statements, 87–90
  entering data into one-dimensional arrays, 424–425
  entering data into two-dimensional arrays, 491–492
  implicit type conversion in, 61, 424
  vs. variable declaration statements, 88
  See also calculation statements
asterisk (*). *See* multiplication operator
asterisk–equal sign (*=): multiplication assignment operator, 97
asterisks program, 275–283
attributes (of objects), **696**, 698
  See also data members (of classes)
average test score program, 400–401
average three numbers problem, 41–43

B

backslash. *See* \ (backslash)
bars (||). *See* Or operator
base 2 number system. *See* binary number system
base 10 number system. *See* decimal number system
base classes, **697**
begin loop comment, 270
behaviors (of objects), **696**, 698
  See also methods (of classes)
binary access files, 583
binary number system, 57, 58
  vs. ASCII codes, 59–60
  vs. decimal number system, 58
binary operators, 83–84
blank character, 79
boldfaced items in statement syntax, 125
bonus calculator program, 340–346
bonus program, 173–175
`bool` data type, 56*+fig*
Boolean data (values), 60
Boolean expressions. *See* logical expressions
Boolean operators. *See* logical operators
bottom-driven loops. *See* posttest loops
braces ({}):
  array *initialValues* delimiters, 422
  code block delimiters, 96, 125, 232
`break` statements (in `switch` statements), **183**, 184
bubble sort program, 454–461
bugs, **92**
  See also debugging…
built-in data types (fundamental data types), **56***+fig*
built-in functions, 309
  sequential access file functions, 586–596
  `string` functions, 526–559, 569*fig*
  value-returning functions, 309, 310–321; `pow` function, **284**–285, 310; `rand` function, **314**–317; `sqrt` function, **310**–311, 312; `time` function, 317; `tolower` function, **140**–141; `toupper` function, **140**–141
  void functions, 371; `srand` function, **317**–318

C

`calcBill` function, 394, 395, 397
`calcGross` methods, 712–714
`calculateArea` method, 706, 708
calculation statements (Treyson Mobley problem), 88–89, 91
calculations programs:

Moonbucks Coffee program, 436–439
switch statement use, 195
calculations with real numbers, 56–57, 89
Caldwell Company orders program, 488–489, 496–497
coding the algorithm, 495–496
passing arrays to functions in, 507–508
calls to functions. *See* function calls
camel case, **53**
car payment program, 348–356
Carroll Cabinets problem, 216
case:
conventions for identifiers, 53; classes, 698
converting characters to uppercase or lowercase, 140–141
See also case sensitivity
case clauses in switch statements, 183, 184
case sensitivity:
of characters, 58, 137
of identifiers, 53
CD collection program, 583–584, 598–601
coding the algorithm, 596–598
"char": pronunciation of, 56*blb,* 140*blb*
char data type, 56*+fig*
char variables: initial value, 62
character delimiters (''), 61
character literal constants, **61**, 137*blb*
characters (character data), **56**, 58, 79
accessing in string variables, 538–540
ASCII. *See* ASCII codes
case sensitivity, 58, 137
comparing variable contents to both versions of a letter, 137–138, 140–141
converting to uppercase or lowercase, 140–141
determining the number of in string variables, 535–537
duplicating in string variables, 557–558
getting from the keyboard. *See* cin statements
inserting in string variables, 554–556
removing from string variables, 548–551
replacing in string variables, 551–553
searching for in string variables, 544–547
See also char variables; character literal constants; strings
cin object, **79***+fig+blb,* 585
cin statements (input statements), 79–81, 82
for array data: one-dimensional arrays, 425–426; two-dimensional arrays, 492–493
for strings: getline function, 527–531, 568–569*+fig;* ignore function, 531–534, 569*+fig*
circle area problem, 66–68
class definitions, 697–700
GrossPay class, 713*fig*
MonthDay class, 710–711*fig*
placement of, 704–705
Salesperson class, 704–705*+fig*
sections, 698, 703–704
Square class, 706–707*+fig*
class statements, **698**, 703
classes, **56**, 525*(2),* **696**
defining. *See* class definitions
inheritance, 697
members. *See* data members; methods; private members; public members
naming, 698
real number classes, 89
close function, 595–596
closing sequential access files, 595–596
cmath file #include directive, 285, 311, 312
code (source code), **93**
indenting in, 8–10*(3)*
reusing, 333, 371, 696
See also instructions; object code
code block delimiters ({}), 96, 125
code blocks: statement blocks, **125**, 129
codesAndRates array, 500, 501, 502, 503–507
coding algorithms, **3**, 28, 52, 78–90, 93
area of a circle problem, 67–68
Caldwell Company orders program, 495–496
car payment program, 351–352
CD collection program, 596–598
electric bill program, 396–397
Falcon Incorporated program, 512
food fat calories and percentage program, 146–147
Hangman game program, 563–564
Hoover College fees program, 100–101
multiplication tables display program, 294
Professor Chang program, 245
rainfall program, 467–468
sales commission program, 191–192
savings calculator program, 286–287
Sweets-4-You program, 605–608
Wilson Company pay rate program, 501
XYZ Company sales program, 427–429
Colfax Sales program, 236–238
colons (::): scope resolution operator, 586, 707
command-line compilers, 94
comment delimiter (//), 95
comments, **95***+blb,* 96
begin loop comment, 270
end statement comments, 125, 183, 221, 232
company name program, 556–558, 559
comparing real numbers, 127*+blb*
comparing variable contents to both versions of a letter, 137–138, 140–141
comparison operators (relational operators), **127**–128
in if statement conditions, 127, 128–132, 140–141
order of precedence, 127, 128, 139
compiler, **5**, 63
command-line compilers, 94
compound conditions (in if statements), 132, 133, 135–140
subconditions, 132; data type in, 134*blb*
using instead of nested structures, 175–177
computer programs. *See* programs
concatenating string variables, 558–559
concatenation operator (+), 558
condition argument (for clause), 232*+blb,* 233, 234, 235, 236, 430, 432
conditions (of control structures): case clauses in switch statements, 183, 184
See also condition argument (for clause); if statement conditions; loop conditions

const keyword, 63
constants. *See* literal constants; named constants
constructors, **707**
  default constructor, **707**, 709
  parameterized constructors, 709–**710**
consuming the character process (getline function), 527
control structures:
  conditions. *See* conditions
  sequence structure, **6**, 23, 120
  See also repetition structures; selection structures
convert dollars program, 390–393
converting characters to uppercase or lowercase, 140–141
counter-controlled pretest loops:
  for statements in, 231–241
  while statements in, 228–231
counters, **224**–228
  initializing and updating, 224–225, 229*+blb*
cout object, **81***+blb,* 585
cout statements (output statements), 81–82
  displaying array contents: one-dimensional arrays, 426–427; two-dimensional arrays, 494–495
  formatting numbers, 141–144
.cpp extension, 93
Creative Sales program, 526, 528–531, 532–534
ctime file #include directive, 317

D

data:
  entering: into one-dimensional arrays, 424–426; into two-dimensional arrays, 491–493
  valid and invalid, **34**
  validation of, 135–136
  writing to sequential access files, 590–592
  See also characters; numbers; strings
data members (of classes), **698**, 699–700
  accessing private members, 707–708
  accessing/referring to public members, 700–701
  initializing private members, 703, 707
  private members with public member methods, 706–709
  public members only, 702–704, 705–706
data types, 52, **53**, 55–57, 59–60
  importance, 60
  return type, 324
  in subconditions, 134*blb*
data validation, 135–136
debugging algorithms: average three numbers problem, 41–43
debugging programs, 92–**93**, 103, 150, 195, 248, 297, 356, 473, 515, 568, 614
  test score program, 401
  See also SWAT THE BUGS exercises
debugging variable declarations, 68
decimal number system, 57
  vs. binary number system, 58
decision making, 120–122, 164–167
  See also selection structures
decision symbol (in flowcharts), **123**, 124*fig*
declaration section (class definitions), **698**, 703–704
declaring and initializing:
  named constants, 63–64, 525
  one-dimensional arrays, 422–424; default values, 423*+blb*
  string variables, 525
  two-dimensional arrays, 489–491
  variables, 62–63, 94–95*+blb. See also* variable declaration statements
declaring memory locations, 62–65
  See also declaring and initializing
default clause (in switch statements), 183, 184, 186
default constructor, **707**, 709
defining classes. *See* class definitions
delimiterCharacter argument:
  getline function, 527
  ignore function, 531
demoted values in type conversion, 61, 86
derived classes, **697**
desk-check tables, **31**–32
  Falcon Incorporated program, 512, 514*fig*
  rainfall program, 470–471*fig*
  XYZ Company sales program, 434–435*+fig*
  See also desk-checking programs
desk-checking algorithms, 24, 31–34, 98–99
  area of a circle problem, 67, 68
  average three numbers problem, 41–43
  bonus program, 174–175, 175–177, 177–178, 178–180
  car payment program, 350
  electric bill program, 395
  food fat calories and percentage program, 146
  gas mileage problem, 35
  Hoover College fees program, 100
  hotel guest bill problem, 38–39
  Miller Incorporated program, 220–221
  multiplication tables display program, 293
  posttest loops, 266–267, 268–269
  pretest loops, 220–221, 266–267
  Professor Chang program, 244
  property tax problem, 40–41
  sales commission program, 191
desk-checking programs, 90–92
  age message program, 378–379, 381–383
  area calculator program, 338–339
  asterisks program, 278–283
  average test score program, 400–401
  bonus calculator program, 343–346
  bubble sort program, 456–461
  calculations program, 195
  car payment program, 352, 355–356
  electric bill program, 398–399
  ending balance program, 355–356
  exam score program, 472–473
  food fat calories and percentage program, 147
  grade message program, 185–186
  Holmes Supply Company program, 234–236
  Hoover College fees program, 101
  incrementing numbers program, 297
  Jasper Music Company program, 230–231
  KL Motors program, 440–441
  multiplication tables display program, 295
  pass/fail display program, 149–150
  Professor Chang program, 246
  random numbers program, 448–452
  salary program, 387–389
  sales commission program, 192
  Sales Express program, 226–228
  squared number program, 248
  total sales program, 612–613

Wilson Company pay rate program, 503–507
XYZ Company sales program, 429–433
desk-checking variable assignments, 103
determining:
the highest number in one-dimensional arrays, 444–452
the number of characters in `string` variables, 535–537
whether a file has been read to the end, 594–595
whether a file was opened successfully, 589–590
Dev C++ IDE, 94
development tools, 94
directives. *See* `#include` directives
`display array` function, 433, 434–435
display date program, 711–712
`displayAge` function, 377, 378, 382–383
`displayArray` functions:
Caldwell Company orders program, 507, 508
random numbers program, 445–446, 447, 448–449
`displayBill` function, 394, 395, 397
`displayCds` function, 583, 584, 597–598
`displayCompanyInfo` function, 372*(2),* 374
displaying array contents:
one-dimensional arrays, 426–427
two-dimensional arrays, 494–495
displaying messages on the screen. *See* `cout` statements
`displayLine` function, 371, 372, 373–374
`displayMonthly` function, 466, 468, 470
`displayTotal` function:
rainfall program, 466, 468, 470
Sweets-4-You program, 604, 607–608
`displayTotalSales` function, 372*(2),* 374
dividing one integer by another, 85–86
division assignment operator (`/=`), 97
division operator (`/`), 83
order of precedence, 83, 139
`do while` statements (loops), 270–272
beginning, 270
displaying array contents: one-dimensional arrays, 426–427; two-dimensional arrays, 494–495
loop condition, 270
syntax, 270, 271
`double` data type, 56*+fig*
vs. `float` data type, 56–57
double quotation marks (""):
#include directive delimiters, 705
string delimiters, 61
`double` variables: initial value (usual), 62
dual-alternative selection structures, **122**, 125, 126, 164–167, **170**
flowcharts, 123–124*+fig,* 168–170*+fig*
nested structures, 166–167, 170–172, 173–175; flowcharts, 168–170*+fig*
the sum of/difference between two real numbers, 130–131
duplicating characters in `string` variables, 557–558

E

e notation, 61*blb,* 141, 142
editors (text editors), 93
development tools, 94
electric bill program, 393–400
`else` clause (in `if` statements), **125**–126*+blb*
See also dual-alternative selection structures
`else if` statements (in `if` statements), 182*fig*
See also multiple-alternative selection structures
`employee` object, 702, 704
employment opportunities for programmers and software engineers, 3
empty string, **61**, 525
encapsulation (in OOP), 696
end of file: testing for, 594–595
end statement comments, 125, 183, 221, 232
ending balance program, 355–356
`endl` stream manipulator, **81**, 591
endless loops, **223**
stopping, 223*+blb,* 248–249*+blb*
entering data:
into one-dimensional arrays, 424–426
into two-dimensional arrays, 491–493
`eof` function, 594–595
equal sign (`=`). *See* assignment operator
equal signs (`==`). *See* equality operator
equality operator (equal to operator) (`==`), 127
vs. assignment operator, 128
order of precedence, 127, 139
`erase` function, 548–551, 569*fig*
escape character (\), 527
escape sequence (\<*character*>), 527
evaluating programs (testing programs), 92–96
car payment program, 352–354
electric bill program, 399
Falcon Incorporated program, 514
food fat calories and percentage program, 148–149
Hangman game program, 567
Hoover College fees program, 101–102
multiplication tables display program, 295–296
Professor Chang program, 246–247
rainfall program, 471–472
sales commission program, 192–193
Sweets-4-You program, 612
See also debugging programs
exam score program, 472–473
exclamation point (`!`): Not operator, 589
exclamation point–equal sign (`!=`). *See* not equal to operator
.exe extension, 93
executable files, **93**
execution of programs: pausing, 96*+blb*
exercises:
introduction to programming, 15–21
memory locations, 73–76
object-oriented programming, 720
one-dimensional arrays, 478–485
problem-solving process, 46–50, 109–118
repetition structures, 254–263, 300–307
selection structures, 154–162, 200–212
sequential access files, 618–625
strings, 574–581
two-dimensional arrays, 517–523
value-returning functions, 362–369
void functions, 406–418

exercises: (*continued*)
See also SWAT THE BUGS exercises; TRY THIS exercises with answers
explicit type conversion, 85–**86**, 87
with the `static_cast` operator, 86–87
exponential notation, 61*blb,* 141, 142
exponentiation, **284**
exposure (in OOP), 697
extended selection structures. *See* multiple-alternative selection structures
extraction operator (>>), **79***+fig+blb,* 526
reading data from sequential access files, 592–593
See also `cin` statements

F

Falcon Incorporated program, 511–514
false path, 8, **122**
flowline, 123, 124*fig*
`fees` array: `types` array and, 461–464
field separator character (#), 591
fields, **590**
file pointer, 587, 588*fig,* 590, 594
filename extensions, 93
files:
code file types, 93
I/O file types, 583, 585
See also sequential access files
`find` function, 544–547, 569*fig*
`fixed` stream manipulator, **141**–142
fixed-point notation, 141–144
`float` data type, 56*+fig*
vs. `double` data type, 56–57
`float` variables: initial value (usual), 62
flowcharts, **28**–29
analyzing. *See* analyzing flowcharts
vs. pseudocode, 29
See also flowcharts for repetition structures; flowcharts for selection structures
flowcharts for repetition structures:
`for` clause, 238*+fig*
nested structures, 277, 278*fig*
posttest loops, 268–269*+fig*
pretest loops, 219–221*+fig,* 267*+fig,* 276*fig*
flowcharts for selection structures:
dual-alternative structures, 123–124*+fig,* 168–170*+fig;* nested structures, 168–170*+fig*
multiple-alternative structures, 181, 188
nested structures, 168–170*+fig*
single-alternative structures, 123–124*+fig*
flowlines, 28–**29**
true and false paths, 123, 124*fig*
food fat calories and percentage program, 145–149, 170–172
`for` clause, 232
arguments, 232–233, 429–430, 431–432
arguments separator, 232*+blb,* 239
flowchart, 238*+fig*
`for` statements (loops), 231–241
in counter-controlled loops, 231–241
displaying array contents: one-dimensional arrays, 426–427; two-dimensional arrays, 494–495
ending, 232
nested: asterisks program, 277–283
searching one-dimensional arrays, 439–442
syntax, 232
formal parameters (of functions), **325**
the address-of operator with, 379–380
in function prototypes, 330, 379–380, 386, 433
list. *See parameterList*
See also actual arguments (of functions)
formatting numeric output, 141–144
forward slash (`/`). *See* division operator
forward slash–equal sign (`/=`): division assignment operator, 97
forward slashes (`//`): comment delimiter, 95
fstream file: `#include` directive, 585
function body, **96**, 324*fig,* 325
function calls:
argumentList, 326, 372
to value-returning functions, 326–330
to void functions, 326, 372, 380
function definitions, 324–325*+fig*
for constructors, 707
placement of, 330
function header, **96**, 324*+fig,* 325
parameter list. *See parameterList*
function names:
creating, 324–325
parentheses in, 140, 315*+blb*
function prototypes, **330**–331
for constructors, 707
formal parameters in, 330, 379–380, 386, 433
functions, **96**
arguments. *See* actual arguments
built-in. *See* built-in functions
calls to. *See* function calls
of classes. *See* methods
definitions. *See* function definitions
naming, 324–325
need for, 309
passing variables to. *See* passing variables to functions
program-defined. *See* program-defined functions
prototypes. *See* function prototypes
receiving. *See* receiving functions
sequential access file functions, 586–596
`string` functions, 526–559, 569*fig*
syntax, 140
types, 96, 309
user-defined. *See* program-defined functions
See also value-returning functions; void functions; *and specific functions*
fundamental data types, **56***+fig*

G

garbage, 62
gas mileage problem, 34–35
generating random numbers (integers), 314–318
initializing the generator, 317–318
from a specific range, 315–317, 333–335
`getAge` function, 380, 381–382
`getBalance` function, 355–356
`getBonus` function, 323, 324*(2),* 325, 342, 345–346
calls to, 327
`getChoice` function, 604, 605
`getHighest` function, 446, 447–448*+blb,* 449–452
`getInput` function, 394*(2),* 396–397
`getline` function, 527–531, 568–569*+fig,* 592–593
`getMonthDay` function, 711*+fig,* 712*fig*
`getNewPayInfo` function, 384–385*(2),* 388–389
`getPayment` function, 349–350, 352

getRandomNumber function, 322–323, 324*(2)*, 325, 331–333, 333–335
  calls to, 326–327, 328–330
getRectangleArea function, 323, 324*(2)*, 325, 337–339, 340
  calls to, 327
getSales function, 341, 344
getSide method, 706, 708*(2)*
getting data from the keyboard. *See* cin statements
getTotal function, 436, 437–439
global variables, **340**
grade message programs, 180–182, 184–186
greater than operator (>), 127
  order of precedence, 127, 139
greater than or equal to operator (>=), 127
  order of precedence, 127, 139
greater than signs (>>). *See* extraction operator
gross pay programs, 135–137, 713–715
GrossPay class, 712
  definition, 713*fig*
  methods, 712–714

H

.h extension, 704
hand-tracing algorithms. *See* desk-checking algorithms
Hangman game program, 562–567
header filename #include directive delimiters, 705*+blb*
header files: including, 704–705
high-level languages, **4**–5
Holmes Supply Company program, 233–236
Hoover College fees program, 99–103
hotel guest bill problem, 37–40
hourly rate program, 442–444
hourlyRates array, 443–444
hypotenuse program, 310–313

I

I/O file objects: creating, 585
I/O file types, 583, 585
I/O symbol (in flowcharts), 28–**29***+fig,* 123, 124*fig*
identifiers (memory location/IPO item names), 52, 53
  case conventions/sensitivity, 53
  selecting, 53–55
IDEs (Integrated Development Environments), **94**
if statement conditions, 7–8, 125
  comparison operators in, 127, 128–132, 140–141
  compound. *See* compound conditions (in if statements)
  logical operators in, 132, 133, 135–140
if statements, **125**–127*+blb*
  conditions. *See* if statement conditions
  else clause, **125**–126*+blb. See also* dual-alternative selection structures
  else if statements, 182*fig. See also* multiple-alternative selection structures
  ending, 125
  exercises, 154–162
  flowcharts. *See* flowcharts for selection structures
  key terms, 151–152
  labs, 144–150, 188–193
  logic errors in. *See* logic errors in selection structures
  nested. *See* nested selection structures
  pseudocode, 173. *See also* logic errors in selection structures; *and specific program IPO charts*
  review questions, 152–154
  summaries, 150–151, 196
  syntax, 126
  See also dual-alternative selection structures; multiple-alternative selection structures; single-alternative selection structures
ifstream class, 585
ignore function, 531–534, 569*+fig*
implementation section (class definitions), **698**, 703–704
implicit type conversion, **61**, 87
  in arithmetic expression processing, 84–85
  in assignment statements, 61, 424
  in declaring arrays, 422
  in entering data into arrays, 424, 491
include directives. *See* #include directives
incrementing counters and accumulators, 224–225
incrementing numbers program, 297
indenting (in code), 8–10*(3)*
inequality operator. *See* not equal to operator
infinite loops. *See* endless loops
inheritance (in OOP), 697
initialization argument (for clause), 232, 233, 234, 235, 429–430, 431–432
initializing:
  counters and accumulators, 224–225, 229*+blb*
  memory locations, 60–61
  private data members, 703, 707
  the random number generator, 317–318
  variables, 61*blb,* 62*(2)+blb;* private data members, 703, 707
  See also declaring and initializing
input, **25**
  identifying, 25
  loop reads, 218–219
  validating, 135–136
  See also cin statements
Input, Processing, and Output charts. *See* IPO charts
input file objects: creating, 585
input files, **583**
input statements. *See* cin statements
input/output symbol (in flowcharts), 28–**29***+fig,* 123, 124*fig*
insert function, 554–556, 569*fig*
inserting characters in string variables, 554–556
insertion operator (<<), **81***+blb*
  writing data to sequential access files, 590–592
  See also cout statements
instantiating objects, 700, 707, 710
instructions (in source code), 94–96
  See also statements
int data type, 56*+fig*
int variables: initial value (usual), 62
integers, **56***+fig*
  determining whether even or odd, 84
  dividing one by another, 85–86
  generating random numbers, 314–318
  swapping higher and lower values, 128–130
  See also average three numbers problem
Integrated Development Environments (IDEs), **94**
internal memory, 52–53
  See also memory locations
interpreter, **5**
introduction to programming, 1–21
  exercises, 15–21
  key terms, 12–13

introduction to programming (*continued*)
lab, 11–12
review questions, 13–15
summary, 12
See also control structures; programmers; programming languages
invalid data, **34**
iomanip file, 142–143
`ios::app` file output mode, 586, 587, 588*fig*
`ios::in` file input mode, 586, 587, 588*fig*
`ios::out` file output mode, 586, 587, 588*fig*
iostream file, 143*blb*
`#include` directive, 585
IPO charts (Input, Processing, and Output charts), **25**–26
See also specific program charts
`is_open` function, 589–590
`istream` class, 585
italicized items in statement syntax, 125
iteration, 9–10

J

Jasper Music Company program, 229–231
Jenko Booksellers program, 498–500
job responsibilities of programmers, 2

K

key terms:
introduction to programming, 12–13
memory locations, 69–70
object-oriented programming, 717–718
one-dimensional arrays, 474
problem-solving process, 44, 105–106
repetition structures, 250, 298
selection structures, 151–152, 196
sequential access files, 614–615
strings, 570
two-dimensional arrays, 516
value-returning functions, 358–359
void functions, 402
keyboard: getting data from. *See* `cin` statements
keywords, **53**
listed, 690
Kindlon High School program, 180–182
KL Motors program, 439–442

L

labs:
answers to, 626–689
introduction to programming, 11–12
memory locations, 65–68
object-oriented programming, 715
one-dimensional arrays, 465–473
problem-solving process, 36–43, 98–103
repetition structures, 241–249, 290–297
selection structures, 144–150, 188–195
sequential access files, 602–614
strings, 560–568
two-dimensional arrays, 509–515
value-returning functions, 347–356
void functions, 390–401
`length` function, 535–537, 569*fig*
less than operator (<), 127
order of precedence, 127, 139
less than or equal to operator (<=), 127
order of precedence, 127, 139
less than signs (<<). *See* insertion operator
letters: comparing variable contents to both versions of a letter, 137–138, 140–141
See also characters
lifetime (of variables), **340**
linker, **93**
literal constants, **61**
character constants, **61**, 137*+blb*
numeric constants, **61***+blb;* named constants vs., 64
string constants, **61**, 137*+blb*
See also named constants
local variables, **129**, **340**
logic errors, **93**
logic errors in selection structures, 173–180
switching the outer and nested decision branches, 177–178
using compound conditions instead of nested structures, 175–177
using unnecessary nested structures, 178–180
logic structures. *See* control structures
logical data (values), 60
logical expressions (Boolean expressions), 125, 127
short-circuit evaluation of, **134**
logical operators (Boolean operators), 132–133
in `if` statement conditions, 132, 133, 135–140
Not operator (`!`), 589
order of precedence, 133, 139
truth tables, **134**–135
See also And operator; Or operator
loop body, **215**, 218
loop conditions, 214
in posttest loops, 265; in `do while` statements, 270
in pretest loops, 215, 218; in `while` statements, 221, 223
sentinel values, 218, 226
loop exit condition, **214**
looping condition, **214**
See also loop conditions
loops, 9–10, 214–263, 265–307
body, **215**, 218
conditions. *See* loop conditions
displaying array contents: one-dimensional arrays, 426–427; two-dimensional arrays, 494–495
endless loops, **223**; stopping, 223*+blb,* 248–249*+blb*
flowcharts. *See* flowcharts for repetition structures
input instructions (reads), 218–219
nested. *See* nested repetition structures
as not required, 216
planning and creating algorithms for, 214–216, 265–267
sentinel values, 218, 226
See also posttest loops; pretest loops
lowercase:
converting characters to, 140–141
vs. uppercase in ASCII, 58
for variables, 53

M

machine language (machine/object code), **4**, **93**
`main` function, 96, 309*(2)*
See also specific programs
making decisions, 120–122, 164–167
See also selection structures
member methods. *See* methods
memory (internal memory), 52–53
See also memory locations
memory location declarations, 62–65
named constants, 63–64

See also variable declaration statements
memory locations, **52**–76
declaring. *See* memory location declarations
exercises, 73–76
initial values, 60–62
initializing, 60–61
key terms, 69–70
labs, 65–68
naming, 53–55. *See also* identifiers
review questions, 71–72
summary, 69
types. *See* data types
See also characters; constants; numbers; strings; variables
messages:
displaying on the screen. *See* `cout` statements
Press any key to continue, 96
methods (of classes), **698**, 699–700
overloaded methods, 712–714
with private data members, 706–709
signature, **710**
Miller Incorporated program:
with `do while` statement, 271–272
with `for` statement, 238–240
with `while` statement, 217–221
mini-quiz answers, 626–689
minus sign (`-`). *See* negation operator; subtraction operator
minus–equal sign (`-=`): subtraction assignment operator, 97
mnemonics, **4**
modifying algorithms, 12, 24
area of a circle problem, 68
hotel guest bill problem, 40
modifying programs:
car payment program, 354
electric bill program, 400
Falcon Incorporated program, 514
food fat calories and percentage program, 149
Hangman game program, 567
Hoover College fees program, 102–103
multiplication tables display program, 296
Professor Chang program, 247–248
rainfall program, 472
random addition problems program, 328–330, 331–333, 333–335
sales commission program, 194
savings calculator program, 287–290
Sweets-4-You program, 612
modulus assignment operator (`%=`), 97
modulus operator (`%`), 83, **84**, 316
order of precedence, 83, 139
`MonthDay` class, 709–710
definition, 710–711*fig*
Moonbucks Coffee program, 436–439
motorcycle membership program, 461–464
movies program, 602–603
multiple-alternative selection structures, **180**–188
flowcharts, 181, 188
grade message program, 180–182
See also `switch` statements
multiplication assignment operator (`*=`), 97
multiplication operator (`*`), 83
order of precedence, 83, 139
multiplication tables display program, 292–297

N

named constants, **53**, 87*blb*
case convention, 53
declaring and initializing, 63–64, 525
vs. numeric literal constants, 53, 64
names. *See* identifiers
namespaces, **96**
naming:
actual arguments, 377
classes, 698
functions, 324–325
I/O file objects, 585
memory locations, 53–55. *See also* identifiers
negation operator (`-`), 83–84
order of precedence, 83, 139
nested repetition structures (loops), 214*blb,* **273**–290
asterisks program, 277–283
`do while` statements, 277
flowchart, 277, 278*fig*
`for` statements, 277–283
savings calculator program, 283–290
nested selection structures (`if` statements), 125, **164**–180
bonus program, 173–175
dual-alternative structures, 166–167, 170–172, 173–175; flowcharts, 168–170*+fig*
flowcharts, 168–170*+fig*
food fat calories and percentage program, 170–172
logic errors in, 173–180
reversing the outer and nested decision branches, 177–178
single-alternative structures, 165–166
using compound conditions instead of, 175–177
using unnecessary structures, 178–180
newline character (\n), 79, 527
writing, 591
not equal to operator (`!=`), 127
order of precedence, 127, 139
Not operator (`!`), 589
number guessing game program, 347–348
number sign (#): field separator character, 591
See also #include directives
number systems, 57–58
numbers (numeric data), 57, 79
binary number system, 57, 58, 59–60
formatting output, 141–144
getting from the keyboard. *See* `cin` statements
incrementing program, 297
promoted/demoted values, 61, 86
square roots: finding, 310–311
squared number program, 248
See also characters; integers; numeric literal constants; real numbers; variables
`numbers` arrays:
bubble sort program, 455, 456–461
random numbers program, 445, 446, 447*+blb,* 448–450
numeric literal constants, **61***+blb*
named constants vs., 53, 64
numeric variables. *See* variables

O

.obj extension, 93
object code (machine code), **4**, **93**
object files, **93**
object-oriented programming, **5**, 695–720
exercises, 720
key terms, 717–718
labs, 715
review questions, 718–720
summary, 716
terminology, 696–697
See also classes; data members; methods; objects

objects, 5*blb,* **696**
I/O file objects, 585
instantiating, 700, 707, 710
`ofstream` class, 585
one-dimensional array elements, 420–421, **422**
accessing, 426–427, 442–444
one-dimensional arrays, 419–485
declaring and initializing, 422–424
determining the highest number in, 444–452
displaying the contents of, 426–427
elements. *See* one-dimensional array elements
entering data into, 424–426
exercises, 478–485
key terms, 474
labs, 465–473
parallel arrays, 461–464
passing to functions, 433–435
as renamed in receiving functions, 434
review questions, 474–477
searching, 439–442
sorting, 454–461
subscripts, **421**
summary, 473–474
`open` function, 586–587, 595
determining the success of, 589–590
opening sequential access files, 586–588
operators:
address-of operator (&), **379**–380, 433
arithmetic. *See* arithmetic operators
assignment operator (=), 62, **87**, 128
comparison. *See* comparison operators
concatenation operator (+), 558
extraction operator (>>), **79***+fig+blb,* 526. *See also* `cin` statements
insertion operator (<<), **81***+blb. See also* `cout` statements
logical. *See* logical operators
Not operator (!), 589
precedence rules. *See* order of precedence rules
scope resolution operator (`::`), 586, 707
`static_cast` operator, **86**–87
unary vs. binary, 83–84
optional item delimiters (`[]`), 423*+blb*
Or operator (`||`), 132, 134, 135, 136*fig,* 137, 138*fig*
order of precedence, 133, 139
order of precedence rules:
arithmetic operators, 83, 128, 139
comparison operators, 127, 128, 139
logical operators, 133, 139
overriding, 83, 84, 127
`orders` array, 487, 488*fig*
`ostream` class, 585
output, **25**
formatting numbers, 141–144
identifying, 25
output file objects: creating, 585
output files, **583**
output statements. *See* `cout` statements
oval (in flowcharts), **28**–29*+fig,* 123, 124*fig*
overloaded methods, 712–714
overriding order of precedence rules, 83, 84, 127

P

parallel one-dimensional arrays, 461–464
parallelogram (in flowcharts), **28**–29*+fig,* 123, 124*fig*
parameterized constructors, 709–**710**
parameterList (formal parameters), 325, 340, 372
in constructors, 710
in overloaded methods, 712–713
parameters. *See* formal parameters
parentheses (`()`):
in function names, 140, 315*+blb*
order of precedence, 83, 139
overriding order of precedence rules with, 83, 84, 127
Pascal case, 698
pass/fail display program, 137–138, 149–150
passing one-dimensional arrays to functions, 433–435
passing two-dimensional arrays to functions, 507–508
passing variables to functions, 326–327, 376–384
by reference, 327, 376, 379–382, 383, 385, 388
by value, 327, 376, 377–379, 382, 385, 388
pausing program execution, 96*+blb*
percent sign (%). *See* modulus operator
percent–equal sign (%=): modulus assignment operator, 97
phone number processing program, 560–562
pi, 53*+blb*
pipe symbol (|), 133
pipe symbols (||). *See* Or operator
planning and creating algorithms, 11, 23–24, 27–31
bonus program, 173
car payment program, 349–350
electric bill program, 393–395
Falcon Incorporated program, 511–513
food fat calories and percentage program, 145–146
gas mileage problem, 35
Hangman game program, 562–566
Hoover College fees program, 99–100
hotel guest bill problem, 37–38
multiplication tables display program, 292–293
Professor Chang program, 243–247
rainfall program, 466–468
for repetition structures, 214–216, 265–267, 273–275
sales commission program, 190–191
for selection structures, 120–122, 164–167
weekly pay problem, 66–67
See also coding algorithms
Plano Elementary School program, 333–335
plus sign (+): concatenation operator, 558
See also addition operator
plus–equal sign (+=): addition assignment operator, 97
polymorphism (in OOP), 697
populating arrays:
one-dimensional arrays, 422–424
two-dimensional arrays, 489–491
posttest loops, **214**, 265–267
desk-checking algorithms, 266–267, 268–269
`do while` statements in, 270–273
flowchart, 268–269*+fig*
loop conditions, 265; in `do while` statements, 270
nested, 277
vs. pretest loops, 265–267
pseudocode, 272. *See also specific program IPO charts*

savings calculator program, 283–290
See also loops
pound sign. *See* number sign (#)
`pounds` array, 436, 437
`pow` function, **284**–285, 310
precedence rules. *See* order of precedence rules
Press any key to continue message, 96
pretest loops, **214**–263
asterisks program, 275–283
counter-controlled loops: `for` statements in, 231–241; with `while` statements, 228–231
counters and accumulators in, 224–228
desk-checking algorithms, 220–221, 266–267
ending: sentinel values for, 218, 226
flowcharts, 219–221*+fig,* 267*+fig,* 276*fig*
`for` statements in, 231–241
labs, 241–249, 290–297
loop conditions, 215, 218; in `while` statements, 221, 223
nested. *See* nested repetition structures
vs. posttest loops, 265–267
problem-solving with, 216–219
pseudocode, 215–216, 217–219. *See also specific program IPO charts*
`while` statements in, 221–231
See also loops
priming read (for loops), **218**
primitive data types (fundamental data types), **56***+fig*
private members (of classes), 699
accessing data members, 707–708
data members with public member methods, 706–709
initializing data members, 703, 707
private variables. *See* under data members; private members
problem-solving process, 22–50, 77–118
analyzing. *See* analyzing problems; analyzing programs
coding. *See* coding algorithms
debugging. *See* debugging algorithms; debugging programs
desk-checking. *See* desk-checking algorithms; desk-checking programs
evaluating. *See* evaluating programs
exercises, 46–50, 109–118
key terms, 44, 105–106
labs, 36–43, 98–103
modifying. *See* modifying algorithms; modifying programs
planning. *See* planning and creating algorithms
review questions, 44–46, 107–108
steps, 23–24, 25–34, 36–43, 78–96, 98–103
summaries, 43, 103–105
problems:
analyzing. *See* analyzing problems
area of a circle, 66–68
average three numbers, 41–43
gas mileage, 34–35
hotel guest bill, 37–40
property tax, 40–41
weekly pay, 36–37, 65–66
See also Treyson Mobley problem
procedure-oriented programming, **5**, 696
process symbol (in flowcharts), 28–**29***+fig,* 123, 124*fig*
processing items (values), **29**–30
product price program, 186–188
Professor Chang program, 243–247
program-defined functions, **309**
ending, 325
value-returning functions, 309, 322–325; calls to, 326–330; `getPayment` function, 349–350, 352; `getSales` function, 341, 344. *See also* `getBonus` function; `getRandomNumber` function; `getRectangleArea` function
void functions, 371–375; calls to, 326, 372, 380; `displayAge` function, 377, 378, 382–383; `displayBill` function, 394, 395, 397; `displayCompanyInfo` function, 372*(2),* 374; `displayLine` function, 371, 372, 373–374; `displayTotalSales` function, 372*(2),* 374; `getAge` function, 380, 381–382; `getInput` function, 394*(2),* 396–397; `getNewPayInfo` function, 384–385*(2),* 388–389
programmers, **2**
employment opportunities, 3
job responsibilities, 2
skills required, 2–3
programming, **2**
introduction to. *See* introduction to programming
object-oriented. *See* object-oriented programming
procedure-oriented programming, **5**, 696
programming languages, **2**
history, 4–6
programs (computer programs), **2**
analyzing. *See* analyzing programs
debugging. *See* debugging programs
desk-checking. *See* desk-checking programs
efficiency: arrays and, 420
evaluating. *See* evaluating programs
pausing execution, 96*+blb*
See also specific programs
promoted values in type conversion, 61, 86
prompts, **81**
Treyson Mobley program, 82*+blb,* 90–91
property tax problem, 40–41
pseudocode, **28**, 30, 35
vs. flowcharts, 29
for `if` statements, 173. *See also* logic errors in selection structures
for posttest loops, 272
for pretest loops, 215–216, 217–219
See also planning and creating algorithms; *and specific program IPO charts*
pseudo-random number generator, 314
initializing, 317–318
public members (of classes), 699
accessing/referring to, 700–701
data members only, 702–704, 705–706
member methods with private data members, 706–709
public variables. *See* under data members; public members

Q

quotation marks:
double. *See* double quotation marks
single (`''`): character delimiters, 61

R

rainfall program, 466–472
`rand` function, **314**–317
RAND_MAX constant, 315
`randNums` array, 444, 445, 448–450
random access files, 583
random addition problems program, 313–321
  modifying, 328–330, 331–333, 333–335
random number generator, 314
  initializing, 317–318
random numbers: generating. *See* generating random numbers
  See also random addition problems program; random numbers program
random numbers program, 444–452
reading data from sequential access files, 592–593
real numbers, **56***+fig*
  calculations with, 56–57, 89
  classes for, 89
  comparing, 127*+blb*
  output formatting, 141–144
  the sum of/difference between two, 130–131
rearranged name program, 544, 546–547
receiving functions, 433
  one-dimensional arrays as renamed in, 434
records, **590**
  separating, 591
  writing to sequential access files, 590–592
rectangle (in flowcharts), **28**–29*+fig,* 123, 124*fig*
referring to public members, 700–701
relational operators. *See* comparison operators
removing characters from `string` variables, 548–551
repetition structures, **9**–10, 213–263, 264–307
  exercises, 254–263, 300–307
  flowcharts. *See* flowcharts for repetition structures
  key terms, 250, 298
  labs, 241–249, 290–297
  planning and creating algorithms for, 214–216, 265–267, 273–275
  review questions, 251–254, 298–300
  summaries, 249–250, 297–298
  See also `do while` statements; `for` statements; loops; posttest loops; pretest loops; `while` statements
`replace` function, 551–553, 569*fig*
replacing characters in `string` variables, 551–553
reserved words. *See* keywords
return data type, 324
`return` statement, **325**, 371
  `return 0`, 96
return value, 309
reusing code, 333, 371, 696
review questions:
  introduction to programming, 13–15
  memory locations, 71–72
  object-oriented programming, 718–720
  one-dimensional arrays, 474–477
  problem-solving process, 44–46, 107–108
  repetition structures, 251–254, 298–300
  selection structures, 152–154, 197–199
  sequential access files, 615–618
  strings, 570–574
  two-dimensional arrays, 516–517
  value-returning functions, 359–362
  void functions, 402–405
reviewing algorithms. *See* desk-checking algorithms
runtime, **53**

S

`salaries` array, 439–441
salary (mean annual wage) for programmers and software engineers, 3
salary program, 384–389
`sales` arrays, 421*+fig,* 429–433, 433, 434
sales commission program, 189–195
Sales Express program, 225–228
sales programs:
  Colfax Sales program, 236–238
  Creative Sales program, 526–534
  Sales Express program, 225–228
  See also total sales programs; XYZ Company sales program
`Salesperson` class, 702, 703–704
  definition, 704–705*+fig*
`saveCd` function, 583, 584, 597
savings calculator program, 283–290
scalar variables, **420**
scientific notation (e notation), 61*blb,* 141, 142
`scientific` stream manipulator, **141**–142
scope (of variables), **340**
scope resolution operator (`::`), 586, 707
screen: displaying data on. *See* `cout` statements
searching:
  arrays: one-dimensional arrays, 439–442; two-dimensional arrays, 500–507
  `string` variables, 544–547
seed actual argument (`srand` function), 317
selection structures, **7**–9, 119–162, 163–212
  determining the need for, 120–122
  dual-alternative. *See* dual-alternative selection structures
  exercises, 154–162, 200–212
  flowcharts. *See* flowcharts for selection structures
  key terms, 151–152, 196
  labs, 144–150, 188–195
  logic errors in. *See* logic errors in selection structures
  multiple-alternative. *See* multiple-alternative selection structures
  nested. *See* nested selection structures
  planning and creating algorithms for, 120–122, 164–167
  review questions, 152–154, 197–199
  single-alternative. *See* single-alternative selection structures
  summaries, 150–151, 196
  See also `if` statements; `switch` statements
selectorExpression (in `switch` statements), 183
semicolon (`;`):
  `for` clause arguments separator, 232*+blb,* 239
  function prototype terminator, 330
  statement terminator, 62
sentinel values (for ending loops), 218, 226
separating records, 591
sequence structure, **6**, 23, 120
sequential access file functions, 586–596

sequential access files, 582–625, **583**
  closing, 595–596
  determining whether opened successfully, 589–590
  determining whether read to the end, 594–595
  exercises, 618–625
  key terms, 614–615
  labs, 602–614
  opening, 586–588
  pointer, 587, 588*fig,* 590, 594
  reading data from, 592–593
  review questions, 615–618
  summary, 614
  writing data to, 590–592
`setMonthDay` function, 711, 712*fig*
`setprecision` stream manipulator, **142**–143
`setSide` method, 706, 707–708*(2)*
shipping charge program, 511–514
`short` data type, 56*+fig*
`short` variables: initial value (usual), 62
short-circuit evaluation (of logical/Boolean expressions), **134**
signature (of a method), **710**
simple variables (scalar variables), **420**
single quotation marks (`''`): character delimiters, 61
single-alternative selection structures, **121**, 125, 126, 173
  flowchart, 123–124*+fig*
  nested structures, 165–166
  swapping higher and lower integer values, 128–130
  See also `if` statements
skills required of programmers, 2–3
slash. *See* forward slash…
social security number program, 553–554, 555–556
software engineers: employment opportunities/salaries (mean annual wages), 3
sorting one-dimensional arrays, 454–461
source code. *See* code
source files, **93**
`sqrt` function, **310**–311, 312
square brackets (`[]`):
  optional item delimiters, 423*+blb*
  subscript delimiters, 421, 422, 487, 507*+blb*
`Square` class, 706
  definition, 706–707*+fig*
square roots: finding, 310–311
squared number program, 248
`srand` function, **317**–318
standard input stream object. *See* `cin` object
standard output stream object. *See* `cout` object
start/stop symbol (in flowcharts), 28–**29***+fig,* 123, 124*fig*
statement blocks, **125**
  variable declaration statements in, 129
statement terminator (`;`), 62
statements, **62**
  class statements, **698**
  end comments, 125, 183, 221, 232
  input. *See* `cin` statements
  output. *See* `cout` statements
  `return` statement, **325**, 371; `return 0`, 96
  syntax formats, 125
  `system("pause");` statement, 96, 379, 383
  See also assignment statements; `cin` statements; `cout` statements; `do while` statements; `for` statements; `if` statements; `switch` statements; variable declaration statements; `while` statements
`static_cast` operator, **86**
  explicit type conversion with, 86–87
stopping endless loops, 223*+blb,* 248–249*+blb*
stream manipulators, **81**
  `endl`, **81**, 591
  `fixed`, **141**–142
  `scientific`, **141**–142
  `setprecision`, **142**–143
stream objects, **79**
streams, **79**
`string` class, 56, 525
  member functions, 526–559, 569*fig. See also specific functions*
`string` data type, 56*+fig,* 525
string delimiters (`""`), 61
string file: `#include` directive, 525
`string` functions, 526–559, 569*fig*
  See also specific functions
string literal constants, **61**, 137*blb*
  empty string, **61**, 525
  See also named constants
`string` variables, 56
  accessing characters in, 538–540
  concatenating, 558–559
  declaring and initializing, 525
  determining the number of characters in, 535–537
  duplicating characters in, 557–558
  initial value, 62
  inserting characters in, 554–556
  removing characters from, 548–551
  replacing characters in, 551–553
  searching for characters in, 544–547
  subscripts, 538
  See also strings
strings (string data/objects), 79, 524–581
  empty string, **61**, 525
  exercises, 574–581
  getting input: `getline` function, 527–531, 568–569*+fig,* 592–593; `ignore` function, 531–534
  key terms, 570
  labs, 560–568
  reading from sequential access files, 592–593
  review questions, 570–574
  summary, 568–569
  See also named constants; string literal constants; `string` variables
student project and test scores (Professor Chang) program, 243–247
subconditions (of compound `if` statement conditions), 132
  data type in, 134*blb*
subject matter experts, 3*blb*
subscript delimiters (`[]`), 421, 422, 487, 507*+blb*
subscripts:
  of one-dimensional arrays, **421**
  of strings, 538
  of two-dimensional arrays, 487
`substr` function, 538–540, 569*fig*
subtraction assignment operator (`-=`), 97
subtraction operator (`-`), 83–84
  order of precedence, 83, 139
sum of/difference between two real numbers display program, 130–131
summaries:
  introduction to programming, 12
  memory locations, 69
  object-oriented programming, 716
  one-dimensional arrays, 473–474
  problem-solving process, 43, 103–105
  repetition structures, 249–250, 297–298

summaries: (*continued*)
selection structures, 150–151, 196
sequential access files, 614
strings, 568–569
two-dimensional arrays, 515
value-returning functions, 357–358
void functions, 401–402
swapping higher and lower integer values program, 128–130
SWAT THE BUGS exercises:
introduction to programming, 19
memory locations, 75
one-dimensional arrays, 479, 483
problem-solving process, 48, 112, 115
repetition structures, 256, 262, 302, 306
selection structures, 155, 159, 201, 208
sequential access files, 619, 622
strings, 575, 580
two-dimensional arrays, 518, 521
value-returning functions, 363, 368
void functions, 407, 415
Sweets Unlimited program, 702–703, 705
Sweets-4-You program, 604, 608–611
`switch` statements, **183**–188
calculations program, 195
`case` clauses in, 183, 184
ending, 183
grade message program, 184–186
lab, 194–195
product price program, 186–187
sales commission program, 194
syntax, 184
syntax, **62**
accessing public members, 700–701
`assign` function, 557*(2)*
class definitions, 698
`close` function, 595, 596
creating I/O file objects, 585*(2)*
declaring and initializing one-dimensional arrays, 422, 423
declaring and initializing two-dimensional arrays, 489, 490
`do while` statements, 270, 271
entering data into one-dimensional arrays: with assignment statements, 424–425; with `cin` statements, 425–426
entering data into two-dimensional arrays: with assignment statements, 491–492; with `cin` statements, 492–493
`eof` function, 595
`erase` function, 548, 549
`find` function, 544–545*(2)*
`for` statements, 232
function prototypes, 330, 331
functions, 140; value-returning functions, 324; void functions, 371
`getline` function, 527, 528, 592–593
`if` statements, 126
`ignore` function, 531, 532
`insert` function, 554, 555
`is_open` function, 589, 590
`length` function, 535, 536
`open` function, 586, 587
reading data from sequential access files, 592, 593
referring to public members, 700–701
`replace` function, 551, 552
`setprecision` stream manipulator, 143
statement formats, 125
`substr` function, 538, 539
`switch` statements, 184
value-returning functions, 324
void functions, 371
`while` statements, 221, 222
writing data to sequential access files, 590, 591
syntax errors, **93**
`system("pause");` statement, 96, 379, 383

T

tab character, 79
test score programs, 400–401, 401
exam score program, 472–473
grade message programs, 180–182, 184–186
testing for end of file, 594–595
testing programs. *See* evaluating programs
text, **52**
text editors. *See* editors
text files. *See* sequential access files
`time` function, 317
`tolower` function, **140**–141
top-driven loops. *See* pretest loops
total and average calculations program, 436–439
total sales programs, 290–292, 465–466, 509–511, 612–613
Moonbucks Coffee program, 436–439
`toupper` function, **140**–141
Treyson Mobley problem (program), 25–34
calculation statements, 88–89, 91
data types, 57
input statements, 80–81, 91
program source code, 94–95
prompts and output statement, 82*+blb,* 90–92
variable declaration statements, 78, 94–95
variable names, 54–55
true path, 8, **122**
flowline, 123, 124*fig*
truth tables, **134**–135
TRY THIS exercises with answers:
introduction to programming, 15–16, 21*ans*
memory locations, 73, 75–76*ans*
one-dimensional arrays, 478, 479, 483–485*ans*
problem-solving process, 46, 49–50*ans,* 109–110, 112, 115–118*ans*
repetition structures, 254–255, 256–257, 263*ans,* 300–301, 302–303, 306–307*ans*
selection structures, 154, 156, 159–162*ans,* 200, 202–203, 209–212*ans*
sequential access files, 618, 619, 623–625*ans*
strings, 574, 575–576, 580–581*ans*
two-dimensional arrays, 517, 518–519, 521–523*ans*
value-returning functions, 362, 364, 368–369*ans*
void functions, 406, 408, 415–418*ans*
two-dimensional array elements, 487
accessing, 494–495
two-dimensional arrays, 486–523, **487**
accumulating values stored in, 498–500
declaring and initializing, 489–491
displaying the contents of, 494–495
elements. *See* two-dimensional array elements
entering data into, 491–493

exercises, 517–523
key term, 516
labs, 509–515
passing to functions, 507–508
review questions, 516–517
searching, 500–507
subscripts, 487
summary, 515
type cast/conversion. *See* explicit type conversion; implicit type conversion
`types` array and `fees` array, 461–464

U

unary operators, 83–84
underscore (_): beginning memory location names with, 54*blb*
update argument (`for` clause), 232, 233, 234, 235, 430, 432
update read (for loops), 218–**219**
updating counters and accumulators, 224–225, 229*+blb*
uppercase:
converting characters to, 140–141
vs. lowercase in ASCII, 58
for named constants, 53
user-defined data types, **56***+fig*
`using` directive, **95**–96
`using namespace std` directive, 96

V

valid data, **34**
validating input, 135–136
value-returning functions, 96, 308–369, **309**, 383
built-in functions, 309, 310–321; `pow` function, **284**–285, 310; `rand` function, **314**–317; `sqrt` function, **310**–311, 312; `time` function, 317; `tolower` function, **140**–141; `toupper` function, **140**–141
calls to, 326–330
exercises, 362–369
key terms, 358–359
labs, 347–356
program-defined functions, 309, 322–325; calls to, 326–330; `getPayment` function, 349–350, 352; `getSales` function, 341, 344. *See also* `getBonus` function; `getRandomNumber` function; `getRectangleArea` function
review questions, 359–362
summary, 357–358
syntax, 324
See also functions
variable declaration statements, 62–63, 94–95*+blb*
assignment statements vs., 88
in statement blocks, 129
variables, **53**, 326, 376
case conventions, 53
`char` variables, 62
in classes. *See* data members
comparing the contents to both versions of a letter, 137–138, 140–141
declaring and initializing, 62–63, 94–95*+blb. See also* variable declaration statements
global variables, **340**
initializing, 61*blb,* 62*(2)+blb;* private data members, 703, 707
lifetime, **340**
local variables, **129**, **340**
passing to functions. *See* passing variables to functions
scalar variables, **420**
scope, **340**
string. *See* `string` variables
verifying:
that data was written correctly to a file, 592
that a file has been read to the end, 594–595
that a file was opened successfully, 589–590
input (validating input), 135–136
void functions, 96, 309, 370–418, **371**, 383
built-in functions, 371; `srand` function, **317**–318
calls to, 326, 372, 380
exercises, 406–418
key terms, 402
labs, 390–401
program-defined functions, 371–375; `calcBill` function, 394, 395, 397; calls to, 326, 372, 380; `displayAge` function, 377, 378, 382–383; `displayBill` function, 394, 395, 397; `displayCompanyInfo` function, 372*(2),* 374; `displayLine` function, 371, 372, 373–374; `displayTotalSales` function, 372*(2),* 374; `getAge` function, 380, 381–382; `getInput` function, 394*(2),* 396–397; `getNewPayInfo` function, 384–385*(2),* 388–389
review questions, 402–405
summary, 401–402
syntax, 371
See also functions
voter eligibility program, 168–170

W

weekly pay problem, 36–37, 65–66
`while` statements (loops), 221–224
in counter-controlled loops, 228–231
counters and accumulators in, 224–228
displaying array contents: one-dimensional arrays, 426–427; two-dimensional arrays, 494–495
ending, 221
loop condition in, 221, 223
syntax, 221, 222
white-space character, **79**
Wilson Company pay rate program, 500–503
coding the algorithm, 501
desk-checking, 503–507
writing data to sequential access files, 590–592
writing the newline character, 591

X

XYZ Company sales program, 421–422
coding the algorithm, 427–429
desk-check table, 434–435*+fig*
desk-checking the code, 429–433
passing arrays to functions in, 433–435

Z

ZIP code program, 535, 536–537, 539–543